Human Errors in Secure Communication Protocols

Master's Thesis
Lara Schmid

Supervisor:
Dr. Saša Radomirović

Professor:
Prof. Dr. David Basin

April 20, 2015

Abstract

We propose a formal model to analyze security protocols with human interaction. We model humans with no knowledge about the protocol and allow an adversary to perform an attack which covers all human errors. We then consider two types of countermeasures. The first type expresses that a human has at least some knowledge about the protocol and will do the known parts correctly. The second type of countermeasures changes a given protocol to be more robust against attacks, such that stronger security properties are achieved with the same assumptions on the human. Further, we introduce hierarchies to compare the human knowledge necessary in different security protocols to achieve the security goals.

Acknowledgements

First I would like to thank Professor David Basin for mentoring me and for giving me the opportunity to do the thesis in his research group. I thank him for finding the time to discuss my thesis and providing me with valuable inputs. I wish to express my special thanks to my supervisor Saša Radomirović for his patient assistance and advice during the last six months. His knowledge and enthusiasm lead to many interesting discussions and helped me a lot. I would also like to express my very great appreciation to my family and friends for their unconditional support. Especially, I want to thank my father for everything he has done for me as well as my mother and my brother, the two stars that are always there for me.

Contents

1 Introduction

People interact with highly sensitive security protocols every day to perform transactions in e-banking or to cast their vote in e-voting protocols. In contrast to protocols where only machines that follow the protocol specifications precisely are communicating with each other, new attacks can arise when interaction with a human has to be taken into account. It is possible that untrained users do not understand the instructions given to them and it is also natural that even experienced humans forget to perform certain steps of a protocol because they are inattentive. Moreover, an adversary can take advantage of the fact that many users cannot differentiate between spoofed and legitimate web pages, for example by tricking them in phishing attacks [13]. If an adversary can get a user to enter sensitive information into a spoofed page, he might be able to learn messages that should have remained secret. Alternatively, he might learn information which he can later use to authenticate himself as the human from whom he has obtained the information.

In this thesis, we introduce a formal model to analyze human errors in security protocols. We abstract away social aspects and reasons behind human errors. This means, for example, that we do not distinguish whether a human left out a step of the protocol due to adversarial influence or because of inattention. We assume that a human has no knowledge about a protocol and that he will follow any instructions given to him over the interfaces he is interacting with. We then model that an adversary is able to control the instructions given to the human by introducing the *role substitution attack*. With this attack, we allow the adversary to choose the actions of a human. He can thus choose what information is sent from the human to whom as well as what information is learned by the human during the protocol execution. Our model is for example able to capture phishing attacks as follows: the human does not know what information he is allowed to give to whom and enters sensitive information into a wrong page because he is asked to do so by the adversary. With the role substitution attack we are also able to cover all other deviations of the human behavior from what he is expected to do by the other participants of the protocol. Consequently, we are able to capture all human errors with this model. We then investigate what countermeasures help to avoid such attacks. For this purpose we introduce two types of countermeasures. The first type allows to express what parts of the protocol the human will perform correctly. The second type of countermeasures makes a given protocol specification more robust in that fewer attacks are possible under the same assumptions on the human's knowledge. We further introduce hierarchies to compare the human knowledge in different protocols.

The thesis is organized as follows. First, we give an overview of related work. Then, we present preliminaries, including the formal definition of protocol specifications and their executions. We next explain in Section 4 our main idea how we model human errors. Further, we introduce countermeasures and define what attacks an adversary can do in the same section. Next, we describe how we model the protocols in the Tamarin tool in Section 5 and introduce countermeasures to make the protocols more robust in Section 6. Afterwards, we present a model extension in Section 7 and then analyze two security protocols in a case study. Finally, we present hierarchies to compare the knowledge of humans in different protocols and give some ideas for future work in sections 9 and 10.

2 Related Work

After a series of accidents in the 1970s and 80s caused by human failures, researchers developed an interest in analyzing and modeling human errors. Among the accidents caused by human errors there were several serious breakdowns of complex systems, including the disaster of Chernobyl. *Rasmussen* [14] states that catastrophic situations are often unique because they result from a combination of failures and inappropriate actions. It is hard to model such situations because control actions are not only dependent on the specific situation but also on the decision making human, his knowledge of the system and how he adapts to the requirements of abnormal situations. In our model, we will also take the human's knowledge of the system into consideration when analyzing security protocols. *Reason* [15] distinguishes two approaches to tackle the human error problem. The *Person approach* focuses on unsafe acts of humans, caused by abnormal mental processes, such as inattention or poor motivation. To reduce these errors, humans are retrained or it is threatened that wrong behavior has certain consequences. In contrast, the *System approach* assumes that human errors are a consequence of properties of the system and can therefore not be avoided by only training the users. To reduce the errors the conditions under which humans work have thus to be improved. We cover two similar aspects in the analysis of security protocols in this work. First, we analyze what has to be known by a human to prevent errors and attacks. Then, we also give ideas how the 'system', here the security protocols, can be made more robust such that less human knowledge is required to achieve the same security goals.
More recent research studies how usable the security is for humans interacting with services over the Internet. *Mannan-Oorschot* [9], for example, analyze what requirements banks assume about their online banking customers when they claim that their service is secure and conclude that most of the users do not meet these requirements. One of the mentioned requirements is that users are expected to install a firewall, anti-virus, and anti-spyware programs before using the services, which can be viewed as the first steps that they are required to do in the protocol. With the model we present, we are able to capture such errors caused by humans that leave away steps of the protocol that they are expected to perform. The authors of [13] examine in usability studies what strategies of phishing can fool users into believing that spoofed websites are legitimate. They conclude that standard indicators designed to signal trustworthiness are not understood by a lot of humans and that more focus should be laid on what humans are able to do and able to recognize. The present thesis models this and other human errors formally, and investigates possible countermeasures that express what knowledge a human has about a protocol. Similar problems, concerning usable security, are discussed at the annually held *Symposium On Usable Privacy and Security (SOUPS)* [1]. The conference covers issues as how to design interfaces usable for humans such that they provide as much security as possible.

Figure 1: Protocol *hisp_codevoting* in Alice&Bob notation

0.	D:	knows($\langle H,S,c,h(c)\rangle$)
0.	S:	knows($\langle H,D,c,h(c)\rangle$)
1.	D $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle S,c,h(c)\rangle/\langle S,c,code\rangle$
2.	H $\circ\!\!\rightarrow\!\!\circ$ P:	*code*
3.	P $\circ\!\!\rightarrow\!\!\circ$ S:	*code/h(c)*

3 Preliminaries

In this section, we first explain an extended Alice&Bob notation we are using to denote security protocols. Afterwards, we define how we normalize these protocols and translate them to formal role specifications. Finally, we define the execution of the protocols as well as the examined security properties.

3.1 Alice&Bob Notation

To describe the security protocols, we use the extended Alice&Bob notation introduced by *Basin et al.* [4]. Thereby, the notation of *Maurer-Schmid* [10] is used to depict the assumptions on the communication channels between two roles. A $\circ\!\!\rightarrow\!\!\circ$ B, A $\bullet\!\!\rightarrow\!\!\circ$ B, A $\circ\!\!\rightarrow\!\!\bullet$ B and A $\bullet\!\!\rightarrow\!\!\bullet$ B denote insecure, authentic, confidential and secure channels from A to B, respectively. This means that a dot indicates the exclusive access of a role to the respective end of the channel while a circle denotes that everybody in the network, including the adversary, has access to it. To denote that a certain pattern is expected to be received by a role, both the message sent and the expected pattern are stated in the notation if they do not match. The two parts are separated by '/'. Finally, *knows()* expresses the initial knowledge of roles and *fresh(m)* or *fresh*($m_1, ..., m_n$) captures that a role generates one or several new fresh values. Separated by the dot '.', *fresh* is placed at the beginning of the same step where the value is sent to the network.

Example 3.1. Figure 1 shows the Alice&Bob notation of the e-voting protocol *hisp_codevoting* which is an adaptation of Chaum's SureVote protocol [5] due to Schläpfer. The participants of the protocol are a human H, his platform P, the voting server S and a code sheet D. We can read from the notation that S initially knows H, the candidate c and the corresponding hash $h(c)$, as well as the code sheet D. Similarly, D contains information on H, S, c and $h(c)$. H has no initial knowledge because no $knows()$ is stated for it. In the first step, the human reads, on a secure channel, from the code sheet the name of the voting server S, his candidate choice c and a corresponding code *code* representing his candidate. While the last of three concatenated messages contained on D consists of the hash value $h(c)$ of c, H interprets and parses this part as one term *code*. The human can then enter the code into his platform P which relays it to the server S, both over insecure channels. Because S knows the hash function, it will parse the term *code* as $h(c)$ again and thus knows what candidate was chosen by the human H.

3.2 Normalizing Alice&Bob Protocols with Human Roles

In this section, we describe normalizing steps we take on a protocol given in Alice&Bob notation to simplify the formalization of the protocol. In particular, we split pairs of messages that are sent over an insecure channel to or from the human such that each of the messages is sent in a separate step, we explicitly mention the generation of fresh values and we enable an interpretation of messages exchanged with the human by adding names to them. In the following paragraphs, we describe each of these normalizing steps in more detail.

Splitting pairs when communicating with a human over insecure channels: We split pairs and longer concatenations of several messages, sent and received by the human over insecure channels into separate messages. We justify this step by the observation that over insecure channels an adversary can change pairs to single messages or vice versa as he wants.

Fresh as separate steps: In the Alice&Bob notation, the generation of fresh values is denoted in the same step in which the value is sent out. We divide the generation and the sending of a fresh value into two separate steps. This does not change the interpretation of the protocol, but explicitly separates two steps that were taken together before.

Example 3.2. The following protocol sends two values from role A to the human role H, the first one being a fresh value.

$$1\,.\quad \text{A} \circ\!\!\rightarrow\!\!\circ \text{H:}\quad fresh(m1).\langle m1, m2\rangle$$

Applying the first two normalizing steps, results in

$$\begin{array}{lrl} 1. & \text{A}: & fresh(m1) \\ 2. & \text{A} \circ\!\!\rightarrow\!\!\circ \text{H:} & m1 \\ 3. & \text{A} \circ\!\!\rightarrow\!\!\circ \text{H:} & m2. \end{array}$$

Adding names to the messages: Normally, a role's interpretation of the messages sent and received in a protocol is given by the role specification. If for example an agent knows from the specification that he will get a session ID in the initialization step of the protocol, he will interpret and parse the first received message as this ID. However, we will assume that humans do not know their role specification, which implies that they do not know the context in which they receive or send a message. As a consequence, we are not able to capture the interpretation of a message by the context of the human role specification. Nevertheless, we need a way to denote messages for defining what the human is allowed to do with them. For this reason, we add to every message a name which denotes its interpretation. Even though names are only necessary in the direct communication with humans, we allow to include them in the messages of all roles to simplify the model.
We adjust the notation and explicitly express each message as a pair consisting of its name and value. We also do this for all messages that are themselves a component of a more complex construction, such as a concatenation or encryption.

Figure 2: Protocol *hisp_codevoting* in Alice&Bob notation after the normalization

$$\begin{array}{rrl}
0. & D: & \text{knows}(\langle\langle \textit{`H',H}\rangle, \langle \textit{`S',S}\rangle, \langle \textit{`candidate',c}\rangle, \langle \textit{`code',h(c)}\rangle\rangle) \\
0. & S: & \text{knows}(\langle\langle \textit{`H',H}\rangle, \langle \textit{`D',D}\rangle, \langle \textit{`candidate',c}\rangle, \langle \textit{`code',h(c)}\rangle\rangle) \\
1. & D \bullet\!\!\rightarrow\!\!\bullet\ H: & \langle\langle \textit{`S',S}\rangle, \langle \textit{`candidate',c}\rangle, \langle \textit{`code',h(c)}\rangle\rangle / \\
 & & \langle\langle \textit{`S',S}\rangle, \langle \textit{`candidate',c}\rangle, \langle \textit{`code',code}\rangle\rangle \\
2. & H \circ\!\!\rightarrow\!\!\circ\ P: & \langle \textit{`code',code}\rangle \\
3. & P \circ\!\!\rightarrow\!\!\circ\ S: & \langle \textit{`code',code}\rangle / \ \langle \textit{`code',h(c)}\rangle
\end{array}$$

Example 3.3. We are given the following protocol where a role A sends two concatenated messages *m1* and *m2*, encrypted with the symmetric key *k(A,B)* to B:

$$1\ .\quad \text{A} \circ\!\!\rightarrow\!\!\circ \text{B} :\quad \{\textit{m1,m2}\}_{k(A,B)}.$$

Adding the names to the protocol results in:

$$1\ .\quad \text{A} \circ\!\!\rightarrow\!\!\circ \text{B} :\quad \{\langle \textit{`message1',m1}\rangle, \langle \textit{`message2',m2}\rangle\}_{k(\langle \textit{`A',A}\rangle, \langle \textit{`B',B}\rangle)}.$$

The only exception where we give one name to a bigger construction of messages is the following. We will define in the model that a human is not able to perform encryption, decryption or to compute hash functions of a value. He is only able to decompose and interpret concatenations of messages or captchas of messages, modeled by the function *captcha*. Consequently, whenever a human receives another constructed term, for example a hash function of a value or an encrypted message, he will only recognize this as one term. For this reason, we introduce one name for such a composite term that is sent to the human to denote its interpretation. Because the names are not relevant in the non-human roles, we then keep this same name for the same term everywhere in the protocol, even in the roles that would be able to decompose it further.

Example 3.4. After applying all normalizing steps, the protocol *hisp_codevoting* from Example 3.1 looks as shown in Figure 2. Recall that the *code* received by the human consists of a hash function h of the candidate c. However, because the human will only recognize it as one term, we give it one name, *'code'*. Consequently, we also name the construct $h(c)$ *'code'* whenever it occurs in the other roles, including for example in the role S where it is recognizable as a hash function.

It is also possible that a non-human role only recognizes a constructed message as one term, for example when it does not have a corresponding key to decrypt an encrypted message. If the same message is not sent to the human in the protocol, it can happen that this term will not have a recognizable name for this role. We do not consider this as an issue, because the non-human roles do not need a name to be able to interpret the messages.

3.3 Protocol Specification

Given a normalized protocol, we define in this section how it can be translated to role specifications.

The role specification language and the definitions we now introduce are closely aligned with the ones of *Cremers-Mauw* [6], with the main difference that we differentiate between human and non-human roles in the protocol.

3.3.1 Role Terms

We define **role terms** as follows:

Definition 1.

$$\begin{aligned} RoleTerm := {} & Var \mid PubVar \mid FreshVar \\ & \mid Func([RoleTerm[, RoleTerm]^*]) \\ & \mid \langle RoleTerm, RoleTerm \rangle \\ & \mid sk(RoleTerm) \mid pk(RoleTerm) \\ & \mid k(RoleTerm, RoleTerm) \\ & \mid \{RoleTerm\}_{RoleTerm} \\ & \mid \langle NaVar, RoleTerm \rangle \end{aligned}$$

We distinguish the basic role terms Var, $PubVar$, and $FreshVar$. Var denotes variable role terms that are placeholders for values to be received during execution. $PubVar$ describes public variables that are known to everybody, including the adversary, while $FreshVar$ is used whenever a role generates a fresh value or already has it in the initial knowledge.
$Func$ is used for function names or constants when used without any arguments. $\langle RoleTerm, RoleTerm \rangle$ denotes the pairing of two role terms or a concatenation when used with more than two arguments. $sk(RoleTerm)$ describes a secret key, $pk(RoleTerm)$ a public key, $k(RoleTerm, RoleTerm)$ a symmetric key and $\{RoleTerm\}_{RoleTerm}$ the encryption of one term with another term, in the subscript, as key. Finally, $\langle NaVar, RoleTerm \rangle$ describes that every role term can be used in a pair together with a variable name $NaVar$. This allows us to add a name to every term sent to the human.
The functions $vars : RoleTerm \to \mathcal{P}(Var)$, $pubvars : RoleTerm \to \mathcal{P}(PubVar)$, $freshvars : RoleTerm \to \mathcal{P}(FreshVar)$, and $varnames : RoleTerm \to \mathcal{P}(NaVar)$ determine the variables, public variables, fresh variables and variable names in a given term, respectively.

The function $()^{-1}$ is defined as the inverse function. In particular, we use it to denote the inverse key of a key.

How non-human roles as well as the adversary can construct and deconstruct messages is described by the **term inference relation** $\vdash$.

Definition 2. *[6] Let M be a set of terms. The* ***term inference relation*** *$\vdash: \mathcal{P}(Term) \times Term$ is defined as the smallest relation satisfying for all terms*

t, t_i, k, and function names f,

$$\begin{aligned}
t \in M &\Rightarrow M \vdash t,\\
M \vdash t_1 \wedge M \vdash t_2 &\Rightarrow M \vdash \langle t_1, t_2 \rangle,\\
M \vdash t \wedge M \vdash k &\Rightarrow M \vdash \{t\}_k,\\
M \vdash \langle t_1, t_2 \rangle &\Rightarrow M \vdash t_1 \wedge M \vdash t_2,\\
M \vdash \{t\}_k \wedge M \vdash k^{-1} &\Rightarrow M \vdash t,\\
\bigwedge_{1 \le i \le n} M \vdash t_i &\Rightarrow M \vdash f(t_1, ..., t_n).
\end{aligned}$$

Because we assume that a human has other capabilities, we distinguish the **human term inference relation** $\vdash_H$ as follows.
A human can neither perform encryption, nor decryption, nor can he compute hash functions, so we do not include these rules. We additionally assume that, unlike non-human roles, a human has the capability to read the contents of a captcha [3], modeled by the function *captcha*. We use *captcha* to model for example graphical challenges that are supposed to be easy readable for a human, but not for a machine. This means that while all non-human roles can compute the function *captcha(x)* of a value x, only the human role can derive x, when given *captcha(x)*.

Definition 3. *We define the* ***human term inference relation*** $\vdash_H : \mathcal{P}(Term) \times Term$ *as the smallest relation satisfying for all terms t, t_1, t_2 and a set of terms M,*

$$\begin{aligned}
t \in M &\Rightarrow M \vdash_H t,\\
M \vdash_H t_1 \wedge M \vdash_H t_2 &\Rightarrow M \vdash_H \langle t_1, t_2 \rangle,\\
M \vdash_H \langle t_1, t_2 \rangle &\Rightarrow M \vdash_H t_1 \wedge M \vdash_H t_2,\\
M \vdash_H captcha(t) &\Rightarrow M \vdash_H t.
\end{aligned}$$

3.3.2 Role Events

In order to define protocols in terms of role specifications, we next define what **role events** can be part of the role specification of role R.

Definition 4.

$RoleEvent_R :=$
$Fresh_R(\langle NaVar, FreshVar\rangle)$
$|In_R([From(\langle NaVar, PubVar\rangle),]RoleTerm)$
$|In\_S_R([From(\langle NaVar, PubVar\rangle),]\langle NaVar, PubVar\rangle, \langle NaVar, R\rangle,$
$RoleTerm)$
$|In\_C_R([From(\langle NaVar, PubVar\rangle),]\langle NaVar, PubVar\rangle, \langle NaVar, R\rangle,$
$RoleTerm)$
$|In\_A_R([From(\langle NaVar, PubVar\rangle),]\langle NaVar, PubVar\rangle, \langle NaVar, R\rangle,$
$RoleTerm)$
$|Out_R([To(\langle NaVar, PubVar\rangle),]RoleTerm)$
$|Out\_S_R([To(\langle NaVar, PubVar\rangle),]\langle NaVar, R\rangle, \langle NaVar, PubVar\rangle, RoleTerm)$
$|Out\_C_R([To(\langle NaVar, PubVar\rangle),]\langle NaVar, R\rangle, \langle NaVar, PubVar\rangle, RoleTerm)$
$|Out\_A_R([To(\langle NaVar, PubVar\rangle),]\langle NaVar, R\rangle, \langle NaVar, PubVar\rangle, RoleTerm)$
$|Claim_R(claim[, \langle NaVar, PubVar\rangle, \langle NaVar, R\rangle], RoleTerm)$

$Fresh$ denotes that the role R generates a fresh value with the value denoted by $FreshVar$ and the name denoted by $NaVar$. In and Out, respectively, describe the receiving and sending of a message, denoted by $RoleTerm$, over an insecure channel. In_S and Out_S are the same for a secure channel, additionally containing pairs $\langle NaVar, PubVar\rangle$ which describe the sender of the message in the second and the receiver of the message in the third argument. Following the same idea, In_C and Out_C as well as In_A and Out_A are the receiving and sending of messages over confidential and authentic channels, respectively. Finally, $Claim$ expresses a belief of the role about a security property that should be given when this event is executed in the protocol. The set $claim$ denotes all possible claims that can be made and depending on the claim one or three arguments follow. Claims are further described in Section 3.5.
The role R that has an event in its specification is depicted by the event's subscript and we define the function $role : RoleEvent \rightarrow Role$ to extract it when given a role event.

The optional first arguments $From$ and To appearing in some of the events denote the role's intention. We want to be able to express a role's belief from whom or to whom he sends a message for later expressing an agreement between roles. Whenever a role sends a message on any channel, the intention *To('R',R)* expresses that the sent message should finally be received by the role R, named *'R'*. Following the same idea, when a role receives a message, the additional argument *From('R',R)* means that it believes that the role R, named *'R'* sent this message.

Example 3.5. The role event $Out\_S_H(To(\langle 'server', S\rangle), \langle 'H', H\rangle, \langle 'D', D\rangle,$ $\langle 'message', m\rangle)$ means that the role H, named *'H'*, sends the message named *'message'* with value m to the role D, with name *'D'*, and intends that the message is finally received by role S named *'server'*. Note that the direct receiver of the message, role D, and the intended end receiver, role S, do not have to match.

The set *RoleEvent* includes the events of all roles:

$$RoleEvent = \bigcup_{R \in PubVar} RoleEvent_R.$$

Additionally, we simplify expressing that a role R, named Rn, sends a message M with the intention $I \in To(\langle NaVar, PubVar\rangle)$ to the role $R2$, named $R2n$, over any channel. These definitions will be useful in the following section where we define the protocol specification. For this purpose, we define the following function returning a set of events:

$$\begin{aligned}
Send_R(I, \langle Rn, R\rangle, \langle R2n, R2\rangle, M) :=\{re \in RoleEvent_R | &re = Out_R(I, M)\\
&\vee re = Out\_S_R(I, \langle Rn, R\rangle, \langle R2n, R2\rangle, M)\\
&\vee re = Out\_C_R(I, \langle Rn, R\rangle, \langle R2n, R2\rangle, M)\\
&\vee re = Out\_A_R(I, \langle Rn, R\rangle, \langle R2n, R2\rangle, M)\}
\end{aligned}$$

Similarly, the following function defines the set capturing the receiving of a message over any channel, with the intention $I \in From(\langle NaVar, PubVar\rangle)$.

$$\begin{aligned}
Receive_R(I, \langle R2n, R2\rangle, \langle Rn, R\rangle, M) :=\{re \in RoleEvent_R | &re = In_R(I, M)\\
&\vee re = In\_S_R(I, \langle R2n, R2\rangle, \langle Rn, R\rangle, M)\\
&\vee re = In\_C_R(I, \langle R2n, R2\rangle, \langle Rn, R\rangle, M)\\
&\vee re = In\_A_R(I, \langle R2n, R2\rangle,\\
&\langle Rn, R\rangle, M)\}
\end{aligned}$$

Note that in both cases it is defined to whom or from whom the message is intended to be sent. If we want to express that we do not care if there is an intention or not, we do not include the first argument. We thus use the same function names as before with one argument less. The resulting set *Send* then includes all events, to any intended role as well as the events that do not include an intended receiver which describe that a message M is sent from the role R, named Rn, to the role *R2*, named *R2n*. *Receive* describes the same if role R

receives a message M from role $R2$. Formally these new sets are defined as

$$\begin{aligned}
Send_R(\langle Rn, R\rangle, \langle R2n, R2\rangle, M) :=\{ & re \in RoleEvent_R | \\
& (\exists i \in \langle NaVar, PubVar\rangle : \\
& re \in Send_R(To(i), \langle Rn, R\rangle, \langle R2n, R2\rangle, M)) \\
& \vee re = Out_R(M) \\
& \vee re = Out\_S_R(\langle Rn, R\rangle, \langle R2n, R2\rangle, M) \\
& \vee re = Out\_C_R(\langle Rn, R\rangle, \langle R2n, R2\rangle, M) \\
& \vee re = Out\_A_R(\langle Rn, R\rangle, \langle R2n, R2\rangle, M)\}
\end{aligned}$$

$$\begin{aligned}
Receive_R(\langle R2n, R2\rangle, \langle Rn, R\rangle, M) :=\{ & re \in RoleEvent_R | \\
& (\exists i \in \langle NaVar, PubVar\rangle : \\
& re \in Receive_R(From(i), \langle R2n, R2\rangle, \\
& \langle Rn, R\rangle, M)) \\
& \vee re = In_R(M) \\
& \vee re = In\_S_R(\langle R2n, R2\rangle, \langle Rn, R\rangle, M) \\
& \vee re = In\_C_R(\langle R2n, R2\rangle, \langle Rn, R\rangle, M) \\
& \vee re = In\_A_R(\langle R2n, R2\rangle, \langle Rn, R\rangle, M)\}.
\end{aligned}$$

Example 3.6. We write $Send_H(To(\langle \text{'}S\text{'}, S\rangle), \langle \text{'}H\text{'}, H\rangle, \langle \text{'}D\text{'}, D\rangle, \langle \text{'}message\text{'}, m\rangle)$ to refer to the set

$$\begin{aligned}
&\{Out_H(To(\text{'}S\text{'}, S), \langle \text{'}message\text{'}, m\rangle), \\
&Out\_S_H(To(\text{'}S\text{'}, S), \langle \text{'}H\text{'}, H\rangle, \langle \text{'}D\text{'}, D\rangle, \langle \text{'}message\text{'}, m\rangle), \\
&Out\_C_H(To(\text{'}S\text{'}, S), \langle \text{'}H\text{'}, H\rangle, \langle \text{'}D\text{'}, D\rangle, \langle \text{'}message\text{'}, m\rangle), \\
&Out\_A_H(To(\text{'}S\text{'}, S), \langle \text{'}H\text{'}, H\rangle, \langle \text{'}D\text{'}, D\rangle, \langle \text{'}message\text{'}, m\rangle)\}.
\end{aligned}$$

The set contains the sending over all possible channels of the message $\langle \text{'}message\text{'}, m\rangle$, sent from role H, named *'H'*, to role D, named *'D'*, and intended for role S, named *'S'*.

3.3.3 Role and Protocol Specification

In this section, we present the role specifications for all the non-human roles, as well as what we call the *original role specification* for the human role. With this, we then define the protocol specification.

A **role specification** for a role R consists of the initial knowledge of R and a sequence of role events.

Definition 5. *The set of all role specifications is given by*

$$\begin{aligned} RoleSpec := \{(m,s) |\ & m \in \mathcal{P}(RoleTerm) \\ & \wedge\ \forall rt \in m : vars(rt) = \emptyset \\ & \wedge\ s \in (RoleEvent_R)^* \\ & \wedge\ wellformed((m,s))\}, \end{aligned}$$

where m denotes the initial knowledge and s the sequence of role events.

We define what it means for a role specification to be well-formed in the following. By $KN_0()$ and $seq()$ we denote the functions that given a role specification return its first component, the initial knowledge m, and its second component, the sequence of events s, respectively.
In order to define well-formedness, we introduce the **accessible subterm relation**, which is defined as follows:

Definition 6. *[6] The* ***accessible subterm relation*** $\sqsubseteq_{acc}$ *is defined as the reflexive, transitive closure of the smallest relation satisfying the following for all terms t_1, t_2: $t_1 \sqsubseteq_{acc} \langle t_1, t_2 \rangle, t_1 \sqsubseteq_{acc} \{t_1\}_{t_2}, t_2 \sqsubseteq_{acc} \langle t_1, t_2 \rangle$.*

The functions $vars$, $pubvars$ and $freshvars$ are generalized to determine all variables, public variables, and fresh variables that occur in a sequence of role events ($RoleEvent^*$). We define **well-formedness** as a predicate on a role specification (m, s).

Definition 7.

$$\begin{aligned} wellformed((m,s)) \Longleftrightarrow & (\forall V \in vars(s) : \exists s', Rn, R, R2n, R2, rt, s'', ev : \\ & s = s' \cdot [ev] \cdot s'' \wedge ev \in Receive_R(\langle R2n, R2 \rangle, \langle Rn, R \rangle, rt) \\ & \wedge V \notin vars(s') \wedge V \sqsubseteq_{acc} rt) \\ & \wedge (\forall F \in freshvars(s) : \\ & (\exists s', R, Fn, s'' : s = s' \cdot [Fresh_R(\langle Fn, F \rangle)] \cdot s'' \\ & \wedge F \notin freshvars(s')) \vee\ F \in m) \end{aligned}$$

We not only require, that all variables first occur in a receive event (first three lines), but we further demand that every fresh variable is first generated or already exists in the initial knowledge (last three lines). Then, the **protocol specification** is defined as follows.

Definition 8. *[6] A* ***protocol specification*** *is a partial function mapping roles to role behaviors. We define the set Protocol of all possible protocol specifications as all partial functions from Role $\nrightarrow$ RoleSpec. For every protocol $P \in$ Protocol and each role $R \in$ Role, P(R) is the role specification of R.*

With this, we have the role specifications for all roles, including a first role specification for the human which we call original human role specification.

Example 3.7. We illustrate the role specifications on the protocol *hisp_codevoting* from Example 3.1.

The role specifications for the protocol *hisp_codevoting*:

$hisp\_codevoting(D) =$
$(\{\langle\langle\text{'}H\text{'},H\rangle, \langle\text{'}S\text{'},S\rangle, \langle\text{'}candidate\text{'},c\rangle, \langle\text{'}code\text{'},h(c)\rangle\rangle\},$
$[\ Out\_S_D(\langle\text{'}D\text{'},D\rangle, \langle\text{'}H\text{'},H\rangle, \langle\langle\text{'}S\text{'},S\rangle, \langle\text{'}candidate\text{'},c\rangle,$
$\langle\text{'}code\text{'},h(c)\rangle\rangle)\ ]\ )$
$hisp\_codevoting(P) =$
$(\{\},$
$[\ In_P(\langle\text{'}code\text{'},code\rangle),$
$Out_P(\langle\text{'}code\text{'},code\rangle)\ ]\ )$
$hisp\_codevoting(S) =$
$(\{\langle\langle\text{'}H\text{'},H\rangle, \langle\text{'}D\text{'},D\rangle, \langle\text{'}candidate\text{'},c\rangle, \langle\text{'}code\text{'},h(c)\rangle\rangle\},$
$[\ In_S(From(\langle\text{'}H\text{'},H\rangle), \langle\text{'}code\text{'},h(c)\rangle)\ ]\ )$
$hisp\_codevoting(H) =$
$(\{\},$
$[\ In\_S_H(\langle\text{'}D\text{'},D\rangle, \langle\text{'}H\text{'},H\rangle, \langle\langle\text{'}S\text{'},S\rangle, \langle\text{'}candidate\text{'},c\rangle, \langle\text{'}code\text{'},code\rangle\rangle),$
$Out_H(To(\text{'}S\text{'},S), \langle\text{'}code\text{'},code\rangle)\ ]\ )$

The derivation from the Alice&Bob specification is straight forward if we examine from the perspective of every role what messages are being sent and received and over what channels. For all specified senders and receivers of the events, we also include the names. Additionally, we include in roles H and S the intention expressing that at sending out *'code'*, role H intends to communicate it to role S and that at receiving *'code'*, role S intends that its original sender is role H. With this, we already specify some goals of the protocol. We will show in Section 3.5 how the security property goals are further specified by also including the respective claim events to the role specifications.

Note that the relative order of the events in the role specifications is adopted from the Alice&Bob notation. It naturally defines an order of the events.

After applying the simplifications introduced in this section, we have a protocol specification and role specifications for all roles participating in the protocol. This includes what we call the original human role specification. We will explain in Section 4 how we allow the adversary to change the original human role specification to model human errors. Next, we define the execution of the protocols.

3.4 Protocol Execution

The definitions introduced in the following are still closely aligned with the ones of *Cremers-Mauw* [6].
Each agent participating in the protocol can execute any role, possibly multiple times. Moreover, several roles can be executed at the same time by the same agent.
We first describe runs as the executions of role specifications and explain how at runtime role terms get instantiated by run terms. Afterwards, we show how the adversary is modeled and finally we explain the operational semantics.

3.4.1 Runs and Instantiation

A run is an execution or a partial execution of a role specification.
In the following, a set RID of run identifiers is assumed, which we use to distinguish the different runs. With this, run terms are defined as follows.

Definition 9.

$$
\begin{aligned}
RunTerm := & FreshVar^{\#RID} \\
& \mid Func([RunTerm[, RunTerm]^*]) \\
& \mid \langle RunTerm, RunTerm \rangle \\
& \mid sk(RunTerm) \mid pk(RunTerm) \\
& \mid k(RunTerm, RunTerm) \\
& \mid \{RunTerm\}_{RunTerm} \\
& \mid AdversaryFresh \\
& \mid \langle NaVar, RunTerm \rangle
\end{aligned}
$$

A fresh value is expressed by $Fresh^{\#RID}$, where the run identifier RID is added to syntactically denote that the fresh value is local to one run. For the runtime, we do not need a notion of variables or public variables anymore because all variables get instantiated by run terms and all public variables by constants. Thereby, constants are expressed by functions of arity zero. The composite run terms can be interpreted in the same way as the corresponding role terms, namely from top to bottom as functions, pairs, secret, public and symmetric keys, and encrypted run terms, respectively. $AdversaryFresh$ is used to denote fresh values generated by the adversary to differentiate them from the ones generated by agents. They are used to model that an adversary can send any message generated by him to the network (see Section 3.4.2 for more details).

A role term is transformed into a run term by means of an instantiation function and the set $Inst$ of instantiations that can be applied is defined as follows.

Definition 10. *The set **Inst** of instantiations that can be applied is defined as:*

$$RID \times (PubVar \nrightarrow Func()) \times (Var \nrightarrow RunTerm).$$

This means that an instantiation consists of the run identifier and two functions describing how public variables are instantiated by constants and how variables are instantiated by run terms. To describe how an instantiation turns role terms into run terms, term instantiation is defined as follows.

Definition 11. *Given an instantiation $inst = (\theta, \rho, \sigma) \in Inst$, a function $f \in Func$, and role terms rt, rt_i, i = 1,...,n such that $pubvars(rt) \subseteq dom(\rho)$ and $vars(rt) \subseteq dom(\sigma)$, we define **instantiation**, as:*
$\langle inst \rangle : RoleTerm \rightarrow RunTerm :$

$$\langle inst\rangle(rt) = \begin{cases} fv^{\#\theta} & if \;\; rt = fv \in FreshVar, \\ \rho(pv) & if \;\; rt = pv \in PubVar, \\ \sigma(v) & if \;\; rt = v \in Var, \\ \langle vn, \langle inst\rangle(rt_1)\rangle & if \;\; rt = \langle vn, rt_1\rangle, vn \in NaVar, \\ f(\langle inst\rangle(rt_1), ..., \langle inst\rangle(rt_n)) & if \;\; rt = f(rt_1, ..., rt_n), \\ \langle\langle inst\rangle(rt_1), \langle inst\rangle(rt_2)\rangle & if \;\; rt = \langle rt_1, rt_2\rangle, \\ \{\langle inst\rangle(rt_1)\}_{\langle inst\rangle(rt_2)} & if \;\; rt = \{rt_1\}_{rt_2}, \\ sk(\langle inst\rangle(rt_1)) & if \;\; rt = sk(rt_1), \\ pk(\langle inst\rangle(rt_1)) & if \;\; rt = pk(rt_1), \\ k(\langle inst\rangle(rt_1), \langle inst\rangle(rt_2)) & if \;\; rt = k(rt_1, rt_2). \end{cases}$$

Next, we introduce the set of all possible runs.

Definition 12. *[6] The set of all possible runs is defined as $Run = Inst \times RoleEvent^*$.*

To define what role term can be instantiated by what run term when receiving a message at execution, the predicate $Match : Inst \times RoleTerm \times RunTerm \times Inst$ is introduced. The first argument denotes the instantiation already done, the second denotes the pattern in the role term, the third the incoming message and the fourth the resulting instantiation.

The set of run terms that are valid values for a variable is defined by the function $type : Var \to \mathcal{P}(RunTerm)$, $type(V) = RunTerm$, which we adopt from the definition of *'No Type Matching'* of *Cremers-Mauw* [6]. This means that a variable can match any term; it can for example be instantiated by a pair or an encrypted message. We point out that it is possible to consider other definitions of 'type' and refer to *Cremers-Mauw* [6] for more examples thereof.
With this, ***Match*** is defined as:

Definition 13. *[6] For all $inst = (\theta, \rho, \sigma)$, $inst' = (\theta', \rho', \sigma') \in Inst$, $pt \in RoleTerm$ and $m \in RunTerm$, the predicate $Match(inst, pt, m, inst')$ holds if and only if $\theta = \theta'$, $\rho = \rho'$ and*

$$\begin{aligned} &\langle inst'\rangle(pt) = m \wedge \\ &\forall v \in dom(\sigma') : \sigma'(v) \in type(v) \wedge \\ &\sigma \subseteq \sigma' \wedge \\ &dom(\sigma') = dom(\sigma) \cup vars(pt). \end{aligned}$$

The first two lines make sure that the instantiation of the pattern is equal to the message m and that the instantiation is well-typed. The third and fourth line then assure that the new assignment is a superset of the old one and that only the variables currently occurring are newly instantiated.

Run event denotes the combination of an instantiation function and a role event.

Definition 14. *We define the* ***set of run events*** *RunEvent as*

$$Inst \times (RoleEvent \cup \{create(R) | R \in PubVar\})$$

In addition to the role events, the event *create* is introduced to denote the creation of a run for a role, which we need for the operational semantics in Section 3.4.3.
To close the definition of runs, we next define the runs that can be created by a protocol P. Recall for this purpose that we use the function $seq()$ to denote the second component of the pair that defines the role specification, that is the sequence of events in a role specification. Additionally, $ran(f)$ is used to denote the range of a function f.

Definition 15. *The function $runsof : Protocol \times Role \rightarrow \mathcal{P}(Run)$ specifies the* ***runs that can be created*** *for a role $R \in dom(P)$ by the protocol P.*

$$\begin{aligned} runsof(P, R) = &\{((\theta, \rho, \emptyset), s) \mid s = seq(P(R)) \wedge \theta \in RID \\ &\wedge dom(\rho) = pubvars(s) \wedge ran(\rho) = Func()\} \end{aligned}$$

The definition states that the runs created by a protocol P consist of all instantiation-sequence pairs where the sequence s denotes a sequence of events of a role specification $P(R)$. Further, the first element in the instantiation is from the set of run identifiers RID. The function ρ, finally, maps public variables to constants, while no variables are instantiated yet.

At last, the **active run identifiers** are defined as follows.

Definition 16. *[6] Given a set of runs F, we define the* ***set of active run identifiers*** *as*

$$runIDs(F) = \{\theta \mid ((\theta, \rho, \sigma), s) \in F\}.$$

For finally defining the execution in terms of the operational semantics rules we also need to specify what an adversary can do, which is introduced next.

3.4.2 Adversary Model

An adversary model similar to a Dolev-Yao adversary [7] that can control the network is assumed. This means that the adversary can learn and block every message sent over the network as well as inject any new message constructed by him. Further, some of the agents can be compromised by the adversary in which case he learns all their knowledge. This allows the adversary to pretend to be one of the compromised agents.
Agent is introduced to denote the set of constants (0-ary functions) that are used for agents at runtime. This set is then partitioned into $Agent_H$ denoting the honest agents and $Agent_C$ denoting the compromised agents. Because agents are not aware of which other agents are compromised or not, it is assumed that they can also initialize or accept communication with compromised agents.
The **initial adversary knowledge** $AKN_0(P)$ for a protocol P is defined in the following.

Definition 17.

$$AKN_0(P) = AdversaryFresh \cup NaVar \cup \bigcup_{\substack{R \in PubVar \\ \rho \in PubVar \rightarrow Agent \\ \rho(R) \in Agent_C}} \{\langle \theta, \rho, \emptyset \rangle(rt) \mid \theta \in RID \wedge rt \in KN_0(R)\}$$

Thereby the initial adversary knowledge is inferred from the protocol description and consists of the fresh terms that can be generated by the adversary, of all variable names, and of the initial knowledge of all compromised agents.
It is then required that whenever an agent expects to receive a variable of a certain type, the adversary is able to infer from his initial knowledge an unbounded number of terms that fit this type.

With this we have all role specifications as well as an adversary model, so next we define the operational semantics of the protocols.

3.4.3 Operational Semantics Rules

To describe the operational semantics for the security protocols, a labeled transition system $(State, RunEvent, \rightarrow, s_0(P))$ is used. Thereby, $State$ denotes the set of possible states. $\rightarrow \in State \times RunEvent \times State$ denotes the transition from one set of states to another set of states, labeled by one of the run events $\in RunEvent$, specified by the second argument. Finally, the last argument $s_0(P)$ captures the initial state of the protocol.

Definition 18. *The **set of possible states** of a network of agents executing roles in a security protocol is defined as*

$$State = \mathcal{P}(RunTerm) \times \mathcal{P}(RunTerm) \times \mathcal{P}(Run).$$

The first and last component of a state denote the adversary knowledge and the (rest of the) runs still to be executed, respectively. Terms that are sent over secure, authentic or confidential channels can be intercepted by an adversary who can store them for later replay, even if he cannot decompose them. The middle component, therefore, describes the terms that have been intercepted by the adversary.

The **initial state** of a protocol is defined to consists of the initial adversary knowledge and two empty sets because no terms have been intercepted and no runs have been activated yet.

Definition 19. *The **initial state** of a protocol is defined as*

$$s_0(P) = \langle\langle AKN_0(P), \emptyset, \emptyset \rangle\rangle.$$

The **transition relation** which describes how to transition from one set of states to another, is then defined by the operational semantics rules depicted in Figures 3 and 4. We use $\boldsymbol{AKN}$ to refer to the current adversary knowledge, $\boldsymbol{AKS}$ to refer to the set of terms the adversary has intercepted on authentic, confidential and secure channels and $\boldsymbol{F}$ to refer to the current set of active runs. In each rule, above the line premises are stated and the rule then expresses that

if these premises are given the transition below the line can take place.
To denote terms sent over a secure, authentic or confidential channel, we respectively use the functions $Sec(\langle \mathit{NaVar,Func()} \rangle, \langle \mathit{NaVar,Func()} \rangle, RunTerm)$, $Auth(\langle \mathit{NaVar,Func()} \rangle, RunTerm)$ and $Conf(\langle \mathit{NaVar,Func()} \rangle, RunTerm)$.

$$\frac{R \in dom(P) \quad ((\theta,\rho,\emptyset),s) \in runsof(P,R) \quad \theta \notin runIDs(F)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{((\theta,\rho,\emptyset),create(R))} \langle\langle AKN, AKS, F \cup \{((\theta,\rho,\emptyset),s)\}\rangle\rangle}\ create$$

$$\frac{(inst,[e]\cdot s) \in F \quad e = Fresh_R(\langle fn, fv\rangle)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN, AKS, (F \setminus \{(inst,[e]\cdot s)\}) \cup \{(inst,s)\}\rangle\rangle}\ fresh$$

$$\frac{(inst,[e]\cdot s) \in F \quad e = Claim_R(cl, \langle pvn, pv\rangle, \langle Rn, R\rangle, rt) \vee e = Claim_R(cl, rt)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN, AKS, (F \setminus \{(inst,[e]\cdot s)\}) \cup \{(inst,s)\}\rangle\rangle}\ claim$$

$$\frac{(inst,[e]\cdot s) \in F \quad e = Out_R(i,rt)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN \cup \{\langle inst\rangle(rt)\}, AKS, (F \setminus \{(inst,[e]\cdot s)\}) \cup \{(inst,s)\}\rangle\rangle}\ out$$

$$\frac{(inst,[e]\cdot s) \in F \quad e = In_R(i,rt) \quad AKN \vdash m \quad Match(inst,rt,m,inst')}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst',e)} \langle\langle AKN, AKS, (F \setminus \{(inst,[e]\cdot s)\}) \cup \{(inst',s)\}\rangle\rangle}\ in$$

Figure 3: Operational semantics rules, part I

$$\frac{(inst, [e]\cdot s) \in F \quad e = Out\_S_R(i, \langle Rn, R\rangle, \langle pvn, pv\rangle, rt)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN, AKS \cup \{\langle inst\rangle(Sec(\langle Rn, R\rangle, \langle pvn, pv\rangle, rt))\}, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst, s)\}\rangle\rangle}\; out\_s$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = In\_S_R(i, \langle pvn, pv\rangle, \langle Rn, R\rangle, rt) \quad AKS \vdash Sec(\langle an, a\rangle, \langle Rn', R'\rangle, m) \quad Match(inst, \langle\langle pvn, pv\rangle, \langle Rn, R\rangle, rt\rangle, \langle\langle an, a\rangle, \langle Rn', R'\rangle, m\rangle, inst')}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst',e)} \langle\langle AKN, AKS, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst', s)\}\rangle\rangle}\; in\_s$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = Out\_A_R(i, \langle Rn, R\rangle, \langle pvn, pv\rangle, rt)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN \cup \{\langle inst\rangle(rt)\}, AKS \cup \{\langle inst\rangle(Auth(\langle Rn, R\rangle, rt))\}, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst, s)\}\rangle\rangle}\; out\_a$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = In\_A_R(i, \langle pvn, pv\rangle, \langle Rn, R\rangle, rt) \quad AKS \vdash Auth(\langle an, a\rangle, m) \quad Match(inst, \langle\langle pvn, pv\rangle, \langle Rn, R\rangle, rt\rangle, \langle\langle an, a\rangle, \langle Rn', R'\rangle, m\rangle, inst')}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst',e)} \langle\langle AKN, AKS, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst', s)\}\rangle\rangle}\; in\_a$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = Out\_C_R(i, \langle Rn, R\rangle, \langle pvn, pv\rangle, rt)}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst,e)} \langle\langle AKN, AKS \cup \{\langle inst\rangle(Conf(\langle pvn, pv\rangle, rt))\}, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst, s)\}\rangle\rangle}\; out\_c$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = In\_C_R(i, \langle pvn, pv\rangle, \langle Rn, R\rangle, rt) \quad AKS \vdash Conf(\langle Rn', R'\rangle, m) \quad Match(inst, \langle\langle pvn, pv\rangle, \langle Rn, R\rangle, rt\rangle, \langle\langle an, a\rangle, \langle Rn', R'\rangle, m\rangle, inst')}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst',e)} \langle\langle AKN, AKS, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst', s)\}\rangle\rangle}\; in\_c$$

$$\frac{(inst, [e]\cdot s) \in F \quad e = In\_C_R(i, \langle pvn, pv\rangle, \langle Rn, R\rangle, rt) \quad AKN \vdash m \quad Match(inst, \langle\langle pvn, pv\rangle, \langle Rn, R\rangle, rt\rangle, \langle\langle an, a\rangle, \langle Rn', R'\rangle, m\rangle, inst')}{\langle\langle AKN, AKS, F\rangle\rangle \xrightarrow{(inst',e)} \langle\langle AKN, AKS, (F \setminus \{(inst, [e]\cdot s)\}) \cup \{(inst', s)\}\rangle\rangle}\; adv\_c$$

Figure 4: Operational semantics rules, part II

The operational semantics rules can be interpreted as follows. A new run can be created for the role R, if R is in the domain of the protocol P and the run is in the set of possible runs of P. Further, the run identifier cannot be already in use by another run. If all these premises are given, the state of the transition system can be updated to now contain the new run.
If there is a run in F where the next event is a $Fresh$ event, the rule $fresh$ can be applied. The run then proceeds to the next step without an update of AKN, AKS or the instantiation.
The premise of the rule *claim* is that the corresponding event is the next in a run of F. Depending on the claims this event has one of the two forms depicted in the rule. In the conclusion, the adversary knowledge and the set of intercepted terms remain the same and the run proceeds to the next step.
In the conclusion of the *out* rule the adversary knowledge gets updated with the new message sent to the network and the run proceeds to the next step. The premises of the *in* rule then require that there is run term in the adversary knowledge AKN that matches the pattern of the expected message. In the conclusion of the rule, the run proceeds to the next step and the instantiation is updated if new variables get bound through the receiving.
When a term is sent out to a secure channel, using the *out_s* rule, the sent term, captured by $Sec()$ can be intercepted by the adversary, for which reason AKS gets updated with it. The rule *in_s* for receiving a message on a secure channel then has the premise that a term $Sec()$ matching the expected pattern can be derived from AKS.
Similarly, if a message is sent on an authentic channel, captured by rule *out_a*, a term $Auth()$ binding the sender to the sent message is stored in AKS. At the same time the adversary can learn the contents of an authenticated message, for which reason, AKN is updated to contain the sent message. The *in_a* rule then requires that AKS contains a term $Auth()$ which matches the expected pattern.
Following the same ideas, if a message is sent to a confidential channel, in rule *out_c*, the corresponding term $Conf()$ is updated in AKS, while for the rule *in_c* it is required that such a term is derivable from AKS. The difference with respect to the authentic channel rules is that the term $Conf()$ binds the receiver to the message and that the message is not learned by the adversary and thus not updated in AKN. Further, the rule *adv_c* allows the adversary to also confidentially send any m inferable from his knowledge to an agent if it matches the pattern expected by the agent.

$AKN(\xi) = AKN_n$ is defined to be the adversary knowledge after the **finite execution** $\xi = [s_0, \alpha_1, s_1, \alpha_2, ..., \alpha_n, s_n]$, where α_i are the run events and $s_i = \langle\langle AKN_i, AKS_i, F_i\rangle\rangle$ the states of the transition system. Note that there exists a unique finite execution $[s_0, \alpha_1, s_1, \alpha_2, ..., \alpha_n, s_n]$ for every finite trace $t = [\alpha_1, \alpha_2, ..., \alpha_n]$ of the transition system.
Finally, $traces(P)$ is the set of all finite traces of protocol P generated by the labeled transition system.

3.5 Security Properties

In this section, we introduce security properties to specify the goals of a protocol and to evaluate whether they hold. We first explain how we capture security properties from the point of view of a particular role and then define secrecy and authentication properties. The way how security properties are claimed and the first claims introduced are again closely aligned with the ideas of *Cremers-Mauw* [6].

3.5.1 Claiming Security Properties

Because there are security properties that hold for one role but not for another role, security properties must be examined from the point of view of a particular role. For this purpose we introduced the role event $Claim_R$ which expresses that role R claims that it believes a certain security property to be satisfied by the protocol. We define in the following for each possible claim what properties have to be fulfilled in the traces of the labeled transition system associated with the protocol such that the claim holds.

Because we assume that it is possible to communicate with compromised agents, properties are often stated conditional on the honesty of communicating agents. Recall that $ran(\rho)$ denotes the range of the function ρ.

Definition 20. *[6] The predicate* ***honest*** *is defined for instantiations as*

$$honest((\theta, \rho, \sigma)) \iff ran(\rho) \subseteq Agent_H.$$

Given an instantiation of a role event, the predicate *honest* is true if the agent executing the event as well as the intended communication partners are honest.

For the next definition, we recall that the function *role* returns the role of a role event.

Definition 21. *[6] The function actor* $: Inst \times RoleEvent \to Agent$ *is defined as*

$$actor((\theta, \rho, \sigma), ev) = \rho(role(ev)).$$

This means that *actor* returns the constant denoting the agent which executes a given run event.

With these functions we define secrecy and authentication properties in the following sections.

3.5.2 Secrecy

A message is said to be secret if the adversary never learns it. We state a secrecy claim as $Claim_R(secret, RoleTerm)$. R denotes the role stating the claim and the second argument denotes the message claimed to be secret. Secrecy is formally defined as follows:

Definition 22. *Let P be a protocol with role $R \in dom(P)$. The secrecy claim event $\gamma = Claim_R(\textbf{secret}, rt)$ is correct if and only if:*

$$\begin{aligned}&\forall t \in traces(P) : \forall ((\theta, \rho, \sigma), \gamma) \in t : \\ &\quad honest((\theta, \rho, \sigma)) \Rightarrow AKN(t) \nvdash \langle \theta, \rho, \sigma \rangle (rt).\end{aligned}$$

The definition states that a secrecy claim in a trace is correct if whenever the claiming agent as well as its communication partners are honest the adversary does not learn the secret in this trace. Note that we do not make any claims of what happens when either the claiming agent or one of its partners is compromised; in such a case the adversary can trivially learn the secret.

3.5.3 Authentication

Authentication claims express that the claiming agent has a certain belief of who is in another role communicating with him. As observed for example by *Lowe* [8], there are different kinds of authentication properties that can be examined. We define in the following different notions of authentication, beginning with the weakest and adding stronger guarantees. The claims for all of them have the form $Claim_R(claim, \langle NaVar,PubVar\rangle,\langle NaVar,R\rangle, RoleTerm)$. Again, R denotes the role performing the claim and the last element is the message with respect to which the claim is made. The argument $\langle NaVar,R\rangle$ additionally captures the name and value of the role R. The first argument denotes the kind of authentication claim that we consider. Finally, the second argument describes the role with respect to whom the claim is made, that is the one that is believed to have performed a certain action.

One of the weakest authentication claims is to require that whenever an agent believes that another agent sent him a message, the other agent has performed an event, which we refer to as **aliveness**.
Cremers-Mauw [6] further differentiate between several kinds of aliveness. We adopt their definition of 'weak aliveness'.

Definition 23. *Let P be a protocol with roles R and R' $\in dom(P)$. The claim event $\gamma = Claim_R(\boldsymbol{alive}, \langle Rn',R'\rangle, \langle Rn,R\rangle, \langle rtn,rtv\rangle)$ is correct if and only if*

$$\begin{aligned}&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow\\ &\quad \exists ev \in t : actor(ev) = \langle inst\rangle(R').\end{aligned}$$

For defining the next authentication property, the function $runidof(inst)$ is introduced. Given an instantiation $inst$, the function returns the first component of it, its run identifier. The domain of $runidof()$ is extended to run events. Further, the notation $e <_t e'$ is used to denote $\exists i, j : \; i < j \wedge t_i = e \wedge t_j = e'$, with $t = t_0, t_1, ..., t_{(n-1)}$ being a sequence and with t_i denoting the $(i+1)th$ element of t. With this, recent aliveness is defined as follows.

Definition 24. *Let P be a protocol with roles R and R' $\in dom(P)$. The claim event $\gamma = Claim_R(\boldsymbol{recent\_alive}, \langle Rn',R'\rangle, \langle Rn,R\rangle, \langle rtn,rtv\rangle)$ is correct if and only if*

$$\begin{aligned}&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow\\ &\quad \exists ev \in t : actor(ev) = \langle inst\rangle(R')\\ &\quad \wedge \exists ev' \in t : runidof(ev') = runidof(inst)\\ &\quad \wedge ev' <_t ev <_t (inst, \gamma).\end{aligned}$$

Additionally to what is given in the definition of aliveness, recent aliveness requires that there exists an event in the claimed partner that takes place between two events in the claiming party. We express that another event (ev') was executed by the claiming party, by ensuring that the run id of this event equals the run id of the claim event. The property guarantees that the partner agent was not only alive, but also within the time that the claiming run took place, which is our notion of **recentness**. Note that this kind of claim can only be done if the claiming agent first sends a message to its communication partner and then expects an answer. Otherwise, there are no two events that can form the 'frame' to define the time in which the partner must have been alive.

The next notion of authentication is adapted from *Basin et al.* [4] and we call it **authentic_hisp**. It states that whenever an agent claims that he received a message authentically from another agent, this other agent indeed communicated this message and the claiming agent indeed received it. Recall the accessible subterm relation (Definition 6), which we use in the following to say that the claimed message can also be sent and received as part of a composed message.

Definition 25. *Let P be a protocol with roles R and R' $\in dom(P)$. The claim event $\gamma = Claim_R(\boldsymbol{authentic\_hisp}, \langle Rn', R' \rangle, \langle Rn, R \rangle, \langle rtn, rtv \rangle)$ is correct if and only if*

$$
\begin{aligned}
&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow \\
&\quad \exists R1n', R1', R1n, R1, rt', rtan, rtav, (inst', \alpha) \in t : actor(inst', \alpha) = \langle inst \rangle (R') \\
&\quad \wedge \alpha \in Send_{R1'}(\langle R1n', R1' \rangle, \langle R1n, R1 \rangle, rt') \\
&\quad \wedge \langle rtan, rtav \rangle \sqsubseteq_{acc} rt' \wedge \langle inst' \rangle (rtav) = \langle inst \rangle (rtv) \\
&\quad \wedge \exists R2n, R2, rtn'', rt'', (inst'', \beta) \in t : runidof(inst'') = runidof(inst) \\
&\quad \wedge \beta \in Receive_R(\langle R2n, R2 \rangle, \langle Rn, R \rangle, rt'') \wedge \langle rtn'', rtv \rangle \sqsubseteq_{acc} rt'' \\
&\quad \wedge (inst', \alpha) <_t (inst'', \beta) <_t (inst, \gamma).
\end{aligned}
$$

The definition states that whenever an agent performing role R claims that it got a message from an agent in role R' and the communication partners are honest, the following is given: first, the agent believed to be in the role R' sends a message from which a term is accessible whose value is instantiated by the same value as the run term in the claim (line two to four). Then, the same agent that is doing the claim (same runID) receives a message from which this term is also accessible. The last line ensures that the order of the events in the trace is as just described and that the claim then happens after the receiving.

The only difference in the next definition is that the communication partners also agree on the name of the message that is being communicated, for which reason we call it **authentic_hisp_withName**.

Definition 26. *Let P be a protocol with roles R and R' $\in dom(P)$. The claim event $\gamma = Claim_R(\boldsymbol{authentic\_hisp\_withName}, \langle Rn', R' \rangle, \langle Rn, R \rangle, \langle rtn, rtv \rangle)$ is* *correct if and only if*

$$\begin{aligned}
&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow \\
&\quad \exists R1n', R1', R1n, R1, rt', rtan, rtav, (inst', \alpha) \in t : actor(inst', \alpha) = \langle inst \rangle (R') \\
&\quad \wedge \alpha \in Send_{R1'}(\langle R1n', R1' \rangle, \langle R1n, R1 \rangle, rt') \\
&\quad \wedge \langle rtan, rtav \rangle \sqsubseteq_{acc} rt' \wedge \langle inst' \rangle (rtav) = \langle inst \rangle (rtv) \\
&\quad \wedge \langle inst' \rangle (rtan) = \langle inst \rangle (rtn) \\
&\quad \wedge \exists R2n, R2, rt'', (inst'', \beta) \in t : runidof(inst'') = runidof(inst) \\
&\quad \wedge \beta \in Receive_R(\langle R2n, R2 \rangle, \langle Rn, R \rangle, rt'') \wedge \langle rtn, rtv \rangle \sqsubseteq_{acc} rt'' \\
&\quad \wedge (inst', \alpha) <_t (inst'', \beta) <_t (inst, \gamma).
\end{aligned}$$

Additionally to what is given in *authentic_hisp*, the fifth and seventh line of this definition ensure that the instantiations of the sent and received message names are both equal to the one in the claim.

The next authentication property not only requires that the claimed partner sent the message specified in the claim, but that the sending and receiving agents further agree on their identities. This means that the sender intends to send the message to the agent that is finally receiving it, and that the receiving agent thinks that the original sender of the message is the agent who really sent it. An agent might communicate a message to another agent indirectly, by sending it to a third agent which forwards it for example. We, therefore, use the intention to express to or from whom a message is intended to be sent or received. This security property describes the classic agreement defined by *Lowe* [8] but restricted to *Send* and *Receive* events.

Definition 27. *Let P be a protocol with roles R and R' $\in$ dom(P). The claim event $\gamma = Claim_R($**authentic_hisp_agreement**$, \langle Rn', R' \rangle, \langle Rn, R \rangle, \langle rtn, rtv \rangle)$ is correct if and only if*

$$\begin{aligned}
&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow \\
&\quad \exists R1n'', R1'', R1n', R1', R1n, R1, rt', rtan, rtav, (inst', \alpha) \in t : \\
&\quad actor(inst', \alpha) = \langle inst \rangle (R') \\
&\quad \wedge \alpha \in Send_{R1'}(To(\langle R1n'', R1'' \rangle), \langle R1n', R1' \rangle, \langle R1n, R1 \rangle, rt') \\
&\quad \wedge \langle rtan, rtav \rangle \sqsubseteq_{acc} rt' \wedge \langle inst' \rangle (rtav) = \langle inst \rangle (rtv) \\
&\quad \wedge \langle inst' \rangle (rtan) = \langle inst \rangle (rtn) \\
&\quad \wedge \langle inst' \rangle (R1'') = \langle inst \rangle (R) \wedge \langle inst' \rangle (R1n'') = \langle inst \rangle (Rn) \\
&\quad \wedge \exists R2n, R2, rt'', (inst'', \beta) \in t : runidof(inst'') = runidof(inst) \\
&\quad \wedge \beta \in Receive_R(From(\langle Rn', R' \rangle), \langle R2n, R2 \rangle, \langle Rn, R \rangle, rt'') \\
&\quad \wedge \langle rtn, rtv \rangle \sqsubseteq_{acc} rt'' \\
&\quad \wedge (inst', \alpha) <_t (inst'', \beta) <_t (inst, \gamma).
\end{aligned}$$

Additionally to what was already required in the former definitions, the agent that is claimed to have sent the message now needs to express an intended receiver $\langle R1n'', R1'' \rangle$ in the send event, whose instantiation equals the agent performing the claim. Further, the receiving event also includes the intended sender of the message which is the same agent, with the same name, with respect to whom the claim is made.

Note that when defining the security properties *alive*, *recent_alive*, *authentic_hisp*, *authentic_hisp_withName* and *authentic_hisp_agreement* in this order, everything that is given in one definition is also given in the subsequent properties. This means that there exists a hierarchy between these authentication properties: if one of the properties holds, all previously defined properties are implied.

The last authentication property we introduce corresponds to the classic agreement from *Lowe* [8]. Unlike *authentic_hisp_agreement* two agents can agree on a message even if it was never involved in a *Send* and *Receive* event from one of them to the other. For this purpose, one agent, say A, has to perform a running claim to an instantiation of a message and to an agent, say B, with whom he wants to have the agreement. We say that the two agents agree on the message if the partner agent B claims that he has the same message, with the same name and value, and wants to share it with the agent A. While this claim of B can directly be expressed in a claim event as we have already seen it, we introduce a new kind of claim to capture the running claim of A.
If an agent in role A, named An, wants to agree upon a message named mn and with value mv to share it with an agent in role B, named Bn, he performs a claim of the form $Claim_A(\textbf{\textit{running}}, \langle An,A\rangle, \langle Bn,B\rangle, \langle mn,mv\rangle)$. With this, we define the agreement of two agents on a message as follows:

Definition 28. *Let P be a protocol with roles R and R' ∈ dom(P). The claim event* $\gamma = Claim_R(\textbf{\textit{agreement\_message}}, \langle Rn',R'\rangle, \langle Rn,R\rangle, \langle rtn,rtv\rangle)$ *is correct if and only if*

$$\begin{aligned}
&\forall t \in traces(P) : \forall inst : (inst, \gamma) \in t \wedge honest(inst) \Rightarrow\\
&\quad \exists R1n', R1', R1n, R1, rtn', rtv', (inst', \alpha) \in t : actor(inst', \alpha) = \langle inst\rangle(R')\\
&\quad \wedge\ \alpha = Claim_{R1'}(running, \langle R1n', R1'\rangle, \langle R1n,R1\rangle, \langle rtn', rtv'\rangle)\\
&\quad \wedge\ \langle inst'\rangle(rtv') = \langle inst\rangle(rtv) \wedge \langle inst'\rangle(rtn') = \langle inst\rangle(rtn)\\
&\quad \wedge\ \langle inst'\rangle(R1) = \langle inst\rangle(R) \wedge \langle inst'\rangle(R1n) = \langle inst\rangle(Rn)\\
&\quad \wedge\ \langle inst'\rangle(R1') = \langle inst\rangle(R') \wedge \langle inst'\rangle(R1n') = \langle inst\rangle(Rn')\\
&\quad \wedge\ (inst', \alpha) <_t (inst, \gamma).
\end{aligned}$$

The definition states that if the claim is made by one agent and the communicating parties are honest, an agreement is given if the claimed partner agent has previously performed a running claim on the same message. Thereby, the instantiations of message name, message value, and of both roles' names and values have to be equal.

Example 3.8. If we want to express that the security property goal of protocol *hisp_codevoting* from Example 3.7 is *authentic_hisp_agreement* claimed by the role S with respect to the human H and the code $h(c)$, we add the corresponding claim to the role specification of S. This results in the following role specification of S:

$$\begin{aligned}&hisp\_codevoting(S) = \\ &\quad(\{\langle\langle \text{'}H\text{'},H\rangle, \langle \text{'}D\text{'},D\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle \text{'}code\text{'},h(c)\rangle\rangle\}, \\ &\quad[\ In_S(From(\langle \text{'}H\text{'},H\rangle), \langle \text{'}code\text{'},h(c)\rangle), \\ &\quad Claim_S(authentic\_hisp\_agreement, \langle \text{'}H\text{'},H\rangle, \langle \text{'}S\text{'},S\rangle, \\ &\quad \langle \text{'}code\text{'},h(c)\rangle)\ ]\ )\end{aligned}$$

Note that we have already included the required intention in role H and role S before. That is, the human H intends to send the *code* to S and role S intends to receive it from H.

We have defined role specifications for all roles, including an original human role specification, as well as the execution and security properties of a protocol. Next, we introduce how an adversary can perform an attack that changes the original human role specification.

4 Human Errors as Role Substitution Attacks

4.1 Human Error and Ignorance

One way to define human errors is to categorize them according to different aspects. It can for example be differentiated whether the human made an error deliberately or because he just forgot a certain step in a stress situation. Without taking social aspects into account, we capture all of these cases by generically defining a human error in a protocol as any deviation of the human from the protocol specification.

Another problem when considering human roles in security protocols is that an untrained human does not follow a role specification, but rather reacts to what he sees on the interfaces he is communicating with. This means in particular that if a human is asked for information, he will often input it to an interface without knowing if this step is part of the protocol he wants to take part in. Further, humans might leak information to anybody because they fail to authenticate their communication partner.

We do not distinguish between ignorance and errors of humans. We model humans that do not know their role specification and examine what properties are still satisfied by the protocol. Thereby, we allow the adversary to choose any role specification for the human that he can substitute for the original one. We call this a role substitution attack. Because we can express with the role substitution attack every possible deviation of the human role specification, we are able to capture with this model all attacks arising from human errors as well as from ignorance.

Example 4.1. To model a human that makes the error of telling a secret to a malicious role, a role substitution attack adds a new event to the human role specification which sends out this secret.

Example 4.2. The human error of forgetting a step in the protocol is modeled by deleting, in a role substitution attack, the corresponding event in the human role specification.

In this section, we first introduce countermeasures to then being able to take them into account in the formal definition of the role substitution attack.

4.2 Countermeasures

In our model, we examine attacks where an adversary can exchange the original human role specification by a new one. If we assume that the human has no knowledge about the protocol, he will accept any of these specifications. In this section, we therefore introduce means to limit the capabilities of an adversary by assuming that the human has at least partial knowledge about the protocol. Because the human only performs role specifications that match with his knowledge, this reduces the number of possible human role specifications that an adversary can substitute.

Example 4.3. Let us consider again the voting protocol *hisp_codevoting*, from Example 3.1. One of the goals of this protocol is that the candidate chosen by the human will remain secret. If the human has no knowledge at all about the protocol, an adversary could just change the human's specification so that the human tells the adversary his candidate's name. If, however, the human knows that he cannot tell his candidate's name to anybody this simple attack is not possible anymore.

In the following, we first introduce an extended Alice&Bob notation for the countermeasures and then explain how the different parts of it can be interpreted. After that, we introduce a formalization of the countermeasures similar to the one of protocols.

4.2.1 Notation for Countermeasures

We specify countermeasures in a notation similar to the extended Alice&Bob notation for protocols. The countermeasures are always interpreted together with the Alice&Bob specification of the protocol. Every occurrence of the variable H in the countermeasures denotes the role of the human whose knowledge is being specified. The rest of the countermeasures is best explained by looking at an example.

Example 4.4. The following shows a possible countermeasure together with the normalized Alice&Bob notation of the protocol *hisp_codevoting* from Example 3.1

0.	D:	knows($\langle\langle$*'H',H*$\rangle, \langle$*'S',S*$\rangle, \langle$*'candidate',c*$\rangle, \langle$*'code',h(c)*$\rangle\rangle$)
0.	S:	knows($\langle\langle$*'H',H*$\rangle, \langle$*'D',D*$\rangle, \langle$*'candidate',c*$\rangle, \langle$*'code',h(c)*$\rangle\rangle$)
1.	D $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle$*'S',S*$\rangle, \langle$*'candidate',c*$\rangle, \langle$*'code',h(c)*$\rangle\rangle$/ $\langle\langle$*'S',S*$\rangle, \langle$*'candidate',c*$\rangle, \langle$*'code',code*$\rangle\rangle$
2.	H $\circ\!\!\rightarrow\!\!\circ$ P:	$\langle$*'code',code*$\rangle$
3.	P $\circ\!\!\rightarrow\!\!\circ$ S:	$\langle$*'code',code*$\rangle$/ $\langle$*'code',h(c)*$\rangle$

NoTell('candidate')
NoGet('candidate')
NoTell('code')
NoOverwrite

R1 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle xn,x\rangle, \langle$*'candidate',c*$\rangle, \langle yn,y\rangle\rangle$	
H $\circ\!\!\rightarrow\!\!\circ$ R2:	$\langle$*'code',code*$\rangle$	*to* : $\langle xn,x\rangle$

The knowledge captured above the line depicts *noDo knowledge* and expresses that the human will never do certain things. Below the line, the *known steps* are indicated, which denote exceptions to the noDo knowledge. That is, the known steps describe that even if certain events are generally prohibited by the noDo knowledge, they can be done in the context described by the known steps. Note that the steps look similar to the Alice&Bob notation of the protocol in that every known step includes sender, receiver, the kind of channel used for communication and the communicated message. Whenever there is a to : or $from$: at the end of a known step, they denote that a send is intended for a specified receiver or a receiving intended to come from a specified sender, respectively. An empty set '{}' above the line in the countermeasure denotes that there is no noDo knowledge, while an empty sequence '[ ]' denotes that there are no known steps.

4.2.2 noDo Knowledge

The noDo knowledge describes the human's knowledge that he will never do certain things throughout the protocol. There are three different kinds of noDo knowledge, namely *NoTell()*, *NoGet()* and *NoOverwrite()*, which we present in the following.

NoTell: The function *NoTell(NaVar)* with a variable name as argument, expresses that the human will not communicate a message with this name. This means for the adversary that he cannot include any send events in the human role specification that contain a message with the specified name.

Example 4.5. The following shows a protocol, where role H gets a secret from role S, together with a countermeasure.

$$\begin{array}{lrl}
0. & \text{S:} & \text{knows}(\langle \textit{'secret',secret}\rangle) \\
1. & \text{S} \bullet\!\!\rightarrow\!\!\bullet \text{ H:} & \langle \textit{'secret',secret}\rangle
\end{array}
\qquad
\frac{NoTell(\textit{'secret'})}{[\,]}$$

The $NoTell$ specifies that the human knows he must not send the message called *'secret'* to anybody. Consequently, a send of such a message cannot appear in the human role specifications that the adversary is allowed to derive.

NoGet: In contrast to *NoTell*, *NoGet(NaVar)* says that a human can never receive a message with the specified name. With this, an adversary cannot add a receiving event of a message containing this name to the human role specification.

Example 4.6. In this protocol, role S gets a secret from role H.

$$\begin{array}{lrl}
0. & \text{H:} & \text{knows}(\langle \textit{'secret',secret}\rangle) \\
1. & \text{H} \bullet\!\!\rightarrow\!\!\bullet \text{ S:} & \langle \textit{'secret',secret}\rangle
\end{array}
\qquad
\frac{NoGet(\textit{'secret'})}{[\,]}$$

The countermeasure expresses that the human knows that he must reject any message named *'secret'* from everybody.

NoOverwrite: *NoOverwrite(NaVar)* expresses that when the human knows the message with the specified name, he always stores it with the same value. That is, he will never accept a new value, overwriting the old one, for a message with the same name. For the adversary this means that if a human already knows a message with a certain name, either because it was in his initial knowledge or because he has already received it in an earlier step, a receive of a message with the same name but with a different value cannot be added to the human role specification.

Example 4.7. In this protocol, role H gets from role S two messages named *'message1'* and *'message2'*, which are in H's initial knowledge.

$$\begin{array}{lrl}
0. & \text{H:} & \text{knows}(\langle\langle\textit{'message1'},\textit{m1}\rangle,\langle\textit{'message2'},\textit{m2}\rangle\rangle) \\
0. & \text{S:} & \text{knows}(\langle\langle\textit{'message1'},\textit{m1}\rangle,\langle\textit{'message2'},\textit{m2}\rangle\rangle) \\
1. & \text{S} \circ\!\!\rightarrow\!\!\circ \text{ H:} & \langle\textit{'message1'},\textit{m1}\rangle \\
2. & \text{S} \circ\!\!\rightarrow\!\!\circ \text{ H:} & \langle\textit{'message2'},\textit{m2}\rangle \\
 & & NoOverwrite(\textit{'message1'}) \\
 & & NoOverwrite(\textit{'message2'}) \\
\cline{3-3}
 & & [\,]
\end{array}$$

The $NoOverwrite$(*'message1'*) means that if the human receives the message named *'message1'*, its value has to match the value of *m1* in the initial knowledge because the messages have the same name which is specified not to be overwritten. Further, an adversary can only add additional receive events of messages called *'message1'* to the human role specification if their values also equal *m1*. The same is true for *'message2'*.

We define $NoTell$, $NoGet$ and $NoOverwrite$ without any arguments to denote the same concepts but with respect to all messages specified in the protocol.

Example 4.8. The following countermeasure expresses the same thing as the countermeasure of Example 4.7. Instead of listing a separate $NoOverwrite()$ for every message, we write $NoOverwrite$ without any arguments to capture both of them.

$$\begin{array}{lrl}
0. & \text{H:} & \text{knows}(\langle\langle\textit{'message1'},\textit{m1}\rangle,\langle\textit{'message2'},\textit{m2}\rangle\rangle) \\
0. & \text{S:} & \text{knows}(\langle\langle\textit{'message1'},\textit{m1}\rangle,\langle\textit{'message2'},\textit{m2}\rangle\rangle) \\
1. & \text{S} \circ\!\!\rightarrow\!\!\circ \text{ H:} & \langle\textit{'message1'},\textit{m1}\rangle \\
2. & \text{S} \circ\!\!\rightarrow\!\!\circ \text{ H:} & \langle\textit{'message2'},\textit{m2}\rangle
\end{array}$$

$$\frac{NoOverwrite}{[\,]}$$

4.2.3 Known Steps

There are protocols where some messages should not be communicated to everybody, but still have to be communicated to dedicated entities for the protocol to work. This section explains how to combine the noDo knowledge with known steps to express that some messages can only be communicated in a certain way.

If the same message name appears in $noTell$ or $noGet$ as well as in a known step, it means that despite the general noDo rule the denoted message can be sent or received in the context described by the known step.

Example 4.9. The following describes a protocol where H sends the message $\langle$*'secret',secret*$\rangle$ to role S.

$$\begin{array}{lrl}
0. & \text{H:} & \text{knows}(\langle\textit{'secret'},\textit{secret}\rangle) \\
1. & \text{H} \bullet\!\!\rightarrow\!\!\bullet \text{ S:} & \langle\textit{'secret'},\textit{secret}\rangle
\end{array}$$

Even though we want the message to remain secret, we cannot generally prohibit H to send it to anybody, because this would include that he cannot send it to S and the protocol would not work anymore. For this reason, we express by a known step in the countermeasure that the human cannot send this message to anybody, except to S over a secure channel. The countermeasure that expresses this is shown in the following and we explain how to interpret it in more detail shortly.

$$\begin{array}{ll} \multicolumn{2}{l}{NoTell(\textit{`secret'})} \\ \hline \text{H} \bullet\!\!\rightarrow\!\!\bullet \text{ S:} & \langle \textit{`secret'}, \textit{secret} \rangle \end{array}$$

Sometimes, we need several known steps that state several exceptions for the protocol to work. Further, we sometimes want to express that an exception can only take place after other events have happened. Therefore, we next present how known steps can be interpreted and what it means if they appear together with other known steps.

Sender, receiver, communicated messages and intention: As in the Alice&Bob notation of the protocol specification, the first part in each step denotes the sender and receiver of the message as well as the communication channel used. We introduce an additional channel ?→? , which expresses that we do not care over what channel a send or receive took place. Because we describe events sent and received by the human, each communicated message is described as a pair of message name and message value. Further, because a human can only recognize these role terms, all sent or received messages consist of either a single message, a captcha or a concatenation of messages and captchas. As already seen in the definition of role events, we sometimes want to express for or from whom a role intends to send or receive a message. In the countermeasures the human's knowledge of an intention is expressed by adding at the end of a known receive or send step a $from$: followed by the intended sender or *to* : followed by the intended receiver, respectively.

Example 4.10. We recall the countermeasure for the protocol *hisp_codevoting* from Example 4.4:

$$\begin{array}{lll} \multicolumn{3}{l}{NoTell(\textit{`candidate'})} \\ \multicolumn{3}{l}{NoGet(\textit{`candidate'})} \\ \multicolumn{3}{l}{NoTell(\textit{`code'})} \\ \multicolumn{3}{l}{NoOverwrite} \\ \hline \text{R1} \bullet\!\!\rightarrow\!\!\bullet \text{ H:} & \langle\langle \textit{xn,x}\rangle, \langle \textit{`candidate',c}\rangle, \langle \textit{yn,y}\rangle\rangle & \\ \text{H} \circ\!\!\rightarrow\!\!\circ \text{ R2:} & \langle \textit{`code',code}\rangle & \textit{to} : \langle \textit{xn,x}\rangle \end{array}$$

The first known step describes an exception to the $NoGet$(*'candidate'*) because the same name is used. It describes that the human H knows he can receive *'candidate'* over a secure channel (denoted by •→•) and in a concatenation of three messages, where the middle one is named *'candidate'*. The second step captures that H knows that, despite the $NoTell$(*'code'*), he can communicate the message ⟨*'code',code*⟩ if he sends it to an insecure channel and intended to the receiver ⟨*xn,x*⟩.

Order of the steps: We say that there is a relative order between the known steps of a countermeasure. That is, a known step means that the described exceptional event can only take place if all the preceding steps in the countermeasure have already happened.

Example 4.11. We add to the observations of the previous example, that the countermeasure describes the following: The human will only send out a message named *'code'* if it is intended to be received by a role $\langle xn,x\rangle$, and if he has previously received a message named *'candidate'* over a secure channel and as the middle one of three messages.

Note that it is possible that some of the known steps do not express an exception by themselves. Rather, they are there to say that the event they describe must already have happened before an exceptional event in a later known step can happen.

Example 4.12. The first known step in the following countermeasure does not express an exception to the $NoTell$(*'message1'*). Nevertheless, it is needed to express that $\langle$*'message1',m1*$\rangle$ can only be sent to a role *R1* over a secure channel, if the human previously received $\langle$*'message2',m2*$\rangle$ from the same role *R1*.

$NoTell$(*'message1'*)	
R1 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle$*'message2',m2*$\rangle$
H $\bullet\!\!\rightarrow\!\!\bullet$ R1:	$\langle$*'message1',m1*$\rangle$

Variables and Values: While a role specification describes concrete events and terms the role expects to communicate, the countermeasures denote what the human knows about the protocol. Thereby, the countermeasures define a set of events that suit this knowledge and that can therefore be part of the role specification. For example, if we express in the countermeasures that a human knows he can only receive a value over a secure channel, but he does not know from whom, we accept any sender in the role specification. Further, we can express in the countermeasures that a human knows he will receive three messages, but only knows the name of the the second one. A concrete event of the role specification which is substituted for this known step, has to contain the specified name in the second message while we allow everything for the name of the other two messages.
We express such variables the human does not know and which can freely be chosen by *placeholder variables.* Whenever there is a variable in a step of the countermeasure, it is a placeholder for a role term describing a role, message or variable name and the human will accept anything in this place, with the following exceptions:

- The human knows that the term H always denotes his own role, so it cannot be changed.
- Whenever there is a variable that already occurred in a previous step or in the initial knowledge, the values of the two variables have to match. This models that a human will perform an implicit comparison.
- The parts of the message in quotes are fixed names, not placeholders, and can therefore not be changed.

This means in particular that different placeholder variables can be substituted with the same role or message.

Example 4.13. Given the countermeasure of Example 4.10, the following sequence of events in not allowed as the human role specification:

$$[In\_S_H(\langle\text{'}A\text{'},A\rangle,\langle\text{'}H\text{'},H\rangle,\langle\langle\text{'}P\text{'},P\rangle,\langle\text{'code'},c\rangle,\langle\text{'}D\text{'},D\rangle\rangle),$$
$$Out_H(To(\text{'}D\text{'},D),\langle\text{'code'},code\rangle)].$$

A send of ⟨*'code',code*⟩ intended to be received by the role ⟨*'D',D*⟩ can only take place as described by the second step in the countermeasure. It can, therefore, only be part of the role specification, if the previous step of the countermeasure precedes it in the role specification. This is not the case, because the only preceding event in the role specification does not match the first step of the countermeasure for two reasons. First, step one in the countermeasure denotes that *'candidate'* is received as the middle of three concatenated messages. In the event, however, the middle message is named *'code'* which does not match the known step because fixed variable names are not allowed to be changed. Secondly, even if this name was the same, the role specification would not be allowed because the role terms substituted for the placeholder variable ⟨*xn,x*⟩ occurring in both known steps of the countermeasure are not the same. While ⟨*xn,x*⟩ from the second step got substituted by ⟨*'D',D*⟩, the first message ⟨*xn,x*⟩ in the first event got substituted by ⟨*'P',P*⟩.

Example 4.14. We can now sum up the interpretation of the countermeasure of Example 4.10 as follows: Because of the $NoGet$(*'candidate'*), a human will never accept a message called *'candidate'*, except if he receives it over a secure channel and as the middle one of three messages. Note that the human does not know from whom he will receive it because the adversary can substitute *R1* with anything since the variable never appears elsewhere in the countermeasure. Further, because of the $NoTell$(*'code'*), the human knows that he will only send a message called *'code'* as specified by the second known step. In particular, this means he only sends ⟨*'code',code*⟩ if it is intended for the agent which he received as the first message in the step where he got the message named *'candidate'*. We see this, because the countermeasure repeats the same term ⟨*xn,x*⟩ such that the adversary can only substitute it by the same value. Finally, a message named *'candidate'* will never be sent out, because there is no known step describing an exception to *NoTell('candidate')*.

Claims: One kind of known step which is interpreted slightly different are the claims. Whenever there is a claim specified in a step of the countermeasure, it means that this claim will only be done in the context described by this known step. That is, as with the known steps that describe an exception to a noDo rule, the human knows that he will only do the claim if he has also done all preceding steps of the countermeasure. If several claims require the same steps to already have happened, we denote them in one step, which means that any of these claims can be done in this context. When an authentication claim is described in a countermeasure step, there are five arguments given which denote the kind of claim, the role with respect to whom the claim is made, the human role H, and the name and value of the message with respect to which the claim is made. Secrecy claims are stated with three arguments, the first one denoting the kind of claim and the others denoting the name and value of the message which is claimed to be secret.

Example 4.15. The following countermeasure denotes that the human will only claim *alive* with respect to a role S and a message ⟨*'message',m*⟩ if he has previously received ⟨*'message',m*⟩ from role S.

{}	
S •→• H:	⟨*'message',m*⟩
Claim	$(alive, S, H,$ *'message'*$, m)$

We described how to specify the human knowledge as countermeasure in a suitable notation as well as how to interpret such a notation. In the next section, we give formal definitions for these countermeasures.

4.2.4 Formalizing the Countermeasures

The introduced countermeasures in extended Alice&Bob notation are a description of what role specifications are allowed to be derived when a human has the specified knowledge. In order to make the two concepts more comparable, we introduce in this section means to capture the countermeasures in a notation closer to the role specifications.

Recall from definitions 5 and 8 that $P(R) = (IK, \rho)$ denotes the role specification for R, where IK is the initial knowledge and ρ a sequence of role events. Similarly, we capture a countermeasure for the human role H as $K(H) = (nDK, \sigma)$. The first argument, nDK, denotes the set of all noDo knowledge rules and σ the sequence of known steps from the countermeasure.
The **noDo rules** are defined as follows:

Definition 29.

$$\begin{aligned} \mathit{NoDoRule} :=& \mathit{NoGet}[(NaVar)] \mid \mathit{NoTell}[(NaVar)] \\ & \mid \mathit{NoOverwrite}[(NaVar)] \end{aligned}$$

Note that the argument of $NoTell$, $NoGet$ and $NoOverwrite$ is optional and that it is of the same type $NaVar$ as the variable names in the role terms. The definition of **known steps** is similar to the one of role events. One exception is that a $Fresh$ never happens as a known step. Additionally, In_Any and Out_Any are newly introduced to denote that a known receive or send step can happen over any kind of channel. Further, we change all the arguments to either be of type ‘known step term’ $KnSTerm$, term name $TermName$, placeholder variable $Placeholder$, or to be the role H.

Definition 30.

$KnownStep :=$
$In_H([From(\langle TermName, Placeholder\rangle),]KnSTerm)$
$\mid In\_S_H([From(\langle TermName, Placeholder\rangle),]\langle TermName, Placeholder\rangle,$
$\langle TermName, H\rangle, KnSTerm)$
$\mid In\_C_H([From(\langle TermName, Placeholder\rangle),]\langle TermName, Placeholder\rangle,$
$\langle TermName, H\rangle, KnSTerm)$
$\mid In\_A_H([From(\langle TermName, Placeholder\rangle),]\langle TermName, Placeholder\rangle,$
$\langle TermName, H\rangle, KnSTerm)$
$\mid In\_Any_H([From(\langle TermName, Placeholder\rangle),]\langle TermName, Placeholder\rangle,$
$\langle TermName, H\rangle, KnSTerm)$
$\mid Out_H([To(\langle TermName, Placeholder\rangle),]KnSTerm)$
$\mid Out\_S_H([To(\langle TermName, Placeholder\rangle),]\langle TermName, H\rangle,$
$\langle TermName, Placeholder\rangle, KnSTerm)$
$\mid Out\_C_H([To(\langle TermName, Placeholder\rangle),]\langle TermName, H\rangle,$
$\langle TermName, Placeholder\rangle, KnSTerm)$
$\mid Out\_A_H([To(\langle TermName, Placeholder\rangle),]\langle TermName, H\rangle,$
$\langle TermName, Placeholder\rangle, KnSTerm)$
$\mid Out\_Any_H([To(\langle TermName, Placeholder\rangle),]\langle TermName, H\rangle,$
$\langle TermName, Placeholder\rangle, KnSTerm)$
$\mid Claim_H(claim[, \langle TermName, Placeholder\rangle, \langle TermName, H\rangle], KnSTerm)$

Because every term in the communication with humans has a name to denote its interpretation, we define the **known step terms** $KnSTerm$ to always consist of a pair denoting the name and value of the term.

Definition 31.

$$\begin{aligned} KnSTerm := &\langle TermName, Placeholder\rangle \\ &\mid \langle KnSTerm, KnSTerm\rangle \\ &\mid captcha(KnSTerm) \end{aligned}$$

with

$$TermName := Placeholder \mid NaVar$$

$\langle NaVar, Placeholder\rangle$ denotes a known step term where the name is already fixed by a variable name, while $\langle Placeholder, Placeholder\rangle$ expresses a term whose name is unknown. The only composite terms recognizable by a human are concatenations and captchas, respectively denoted by $\langle KnSTerm, KnSTerm\rangle$ and $captcha(KnSTerm)$.

We can capture the channels $?\rightarrow?$ from the countermeasure in extended Alice&Bob notation by either an *Out_Any* step when the channel is outgoing or by an *In_Any* step if it is incoming from the human's perspective. All the other steps are translated to the formal countermeasure by first choosing the

KnownStep according to the type of channel and whether the message is incoming or outgoing from the human's perspective. The arguments are then read as follows: if there is an intention specified in the extended Alice&Bob notation, this is put into the first argument, otherwise the first argument is omitted. The remaining arguments then denote sender, receiver and the communicated message, in this order. One thing which is not specified in the countermeasures in extended Alice&Bob notation are the names of the sender and receiver, which we express by introducing fresh placeholder variables. The component describing the value of the sender and receiver, as well as of the communicated message can again be copied from the respective terms in the extended Alice&Bob notation. Finally, for every claim in the countermeasure, a corresponding known step is added to the formal countermeasure. Again, because the names of the roles are not specified in the extended Alice&Bob notation, we introduce fresh placeholder variables to express them.

Example 4.16. In this formalism, the countermeasure from Example 4.10 for the protocol *hisp_codevoting* is expressed as follows.

$$\begin{aligned}
&K(H) = \\
&\quad (\{NoGet(\text{'}candidate\text{'}), NoTell(\text{'}candidate\text{'}), NoTell(\text{'}code\text{'}), \\
&\quad NoOverwrite\}, \\
&\quad [In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle), \\
&\quad Out_H(To(\langle xn,x\rangle), \langle \text{'}code\text{'},code\rangle)\ ]\ )
\end{aligned}$$

Unlike in the role specifications, the set in the beginning captures the noDo knowledge. The second part consists of the sequence of all known steps, whose arguments are read from the corresponding countermeasure steps in the extended Alice&Bob notation. Note that the sender and receiver of a message on an insecure channel are not captured because the adversary can always change them.

4.3 Role Substitution Attacks

We now define how a role substitution attack can change the original human role specification, taking into account the knowledge of the human captured in the countermeasures. The non-human role specification will remain as previously described.

We first define which events are accepted by the human for a given known step. We then define what role specifications an adversary is allowed to derive when given the original human role specification and a countermeasure describing the human's knowledge. Given the set of derivable human role specifications, we then define the role substitution attack.
Because the set of possible human role specifications is infinite, we finally define extra conditions to bound the set and to ensure decidable verification.

4.3.1 Accepted Role Events

Recall that a countermeasure for the human H is a pair $K(H) = (nDK, \sigma)$, where the last element σ is a sequence of known steps. These known steps are closely related but not equal to role events: they define what a role event has to fulfill to be 'accepted' as this known step. For this reason, we define which concrete role events can be substituted by an adversary for the known steps in the countermeasures.

First, we introduce some functions to access the contents of known steps. When given a $KnownStep$, the function $direction : KnownStep \rightarrow \{in, out\}$ returns whether the known step is incoming (in) or outgoing (out) from the human's perspective. That is, for all receiving steps the function returns in and for all sending steps out. The function $channel : KnownStep \rightarrow \mathcal{P}(\{insecure, secure, authentic, confidential\})$ returns a set of channel types the event is allowed to be conducted on. For example, when a known step describes a send or receive event over a secure channel, by Out_S or In_S, the function $channel$ returns $\{secure\}$. For steps that have to happen over insecure, authentic and confidential channels, the function returns $\{insecure\}$, $\{authentic\}$ and $\{confidential\}$, respectively. For In_Any and Out_Any steps that can be done over any channel, the function returns the set of all four channels. We extract whether an intention is specified by the function $intent : KnownStep \rightarrow \{1, 0\}$. The function returns 1 if there is an intention $From(Xn, X)$ or $To(Xn, X)$ in a given known step and 0 otherwise. Further, we define $form : KnownStep \rightarrow (\{concatenation, captcha\} \times Integer)^*$ which returns a sequence of pairs denoting the form of the message in the step as follows. The first component of the first pair determines whether the communicated message is a concatenation or a captcha. The second component of the first pair is an integer denoting how many arguments are included in the concatenation or captcha. We say that the form of a captcha is uniquely defined by the amount of its arguments. The form of a concatenation, however, is further distinguished by defining for every argument if it is a single message or a captcha. For this reason, whenever the outer form is a concatenation, the subsequent pairs returned by $form$ denote the form and amount of arguments for every component of the concatenation. Named terms that cannot further be decomposed are considered as a concatenation of one term, so the function returns for example [(*concatenation*, *1*)] for the message ⟨*'message'*,*m*⟩. This means for example that if the message has the form $\langle x, captcha(y), z\rangle$, the function returns [(*concatenation*, *3*), (*concatenation*, *1*), (*captcha*, *1*),(*concatenation*, *1*)] and for the message $captcha(x, y)$ it returns [(*captcha*, *2*)]. Next, we introduce the function $components : KnownStep \rightarrow (Placeholder \cup NaVar)^*$, which given a known step extracts all the components of the known step terms in it. That is, the name and value parts of every pair denoting a known step term are returned in a sequence in the same order as they appear in the step. Stated differently, every occurring placeholder variable and variable name is added to the sequence, respecting their order when the known step is read from left to right. When given a $KnownStep$, the function $extractName : KnownStep \rightarrow (Placeholder \cup NaVar)^*$, returns the sequence of the first components of all occurring terms, denoting the term names, in the order that they appear in the step. This means that this function returns every other element of the sequence returned by $components$. We use the functions $extractName$ and $components$ with a second argument i to denote the $(i+1)th$ element of the returned sequence.
The only kind of known steps that cannot fully be captured with these functions are the claims. To be able to still include them in the same definition, we de-

fine that when given a known step containing a claim, the functions *direction*, *channel* and *intent* always return *in*, $\{insecure\}$ and 0, respectively. Further, $form$ always returns $(concatenation, 1)$, because we always make a claim with respect to one message. The functions *components* and *extractName* still do the same thing and do not take into account the additional first argument of a claim, the kind of claim made. Finally, we introduce the function $claimSort : KnownStep \to claim \cup \{\}$, which returns the kind of claim made when given a known step with a claim as input and the empty set for any other known step.

Example 4.17. We illustrate these functions on the countermeasure from Example 4.16.

$direction(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) = in$
$channel(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) = secure$
$intent(Out_H(To(\langle xn,x\rangle), \langle \text{'}code\text{'},code\rangle)) = 1$
$intent(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) = 0$
$form(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) =$
$[(concatenation, 3), (concatenation, 1), (concatenation, 1), (concatenation, 1)]$
$form(Out_H(To(\langle xn,x\rangle), \langle \text{'}code\text{'},code\rangle)) = [(concatenation, 1)]$
$components(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) =$
$[R1n, R1, Hn, H, xn, x, \text{'}candidate\text{'}, c, yn, y]$
$components(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle), 3) = H$
$extractName(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)) =$
$[R1n, Hn, xn, \text{'}candidate\text{'}, yn]$
$extractName(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle), 3) =$
$\text{'}candidate\text{'}$
$claimSort(In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle \text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle), 3) = \{\}$

Even though role events and known steps are not equal, they are similar enough that we can generalize all these functions to also take *RoleEvents* as inputs. Because the human can only recognize concatenations and captchas as composed role terms, it is enough to distinguish these two cases with the function $form$, even when taking events as inputs. The only functions which will have a different range when given an event as input, are *extractName* which will return values of type $NaVar^*$ and the function *components* which will map role events to a sequence of occurring role terms $\in (Var \cup PubVar \cup FreshVar \cup NaVar)^*$. Conversely, the functions *role*, which returns the role of an event, and *varnames*, which returns all terms of kind $NaVar$, are generalized to also take known steps as input.

Next, we define for every known step a set *accepted* containing all the role events that an adversary can substitute for this known step.

Definition 32. *We say that an event ev ∈ RoleEvent is* ***accepted*** *for a known* *step kns* $\in$ *KnownStep if the following is given:*

$$\begin{aligned}
ev \in accepted(kns) \iff & direction(ev) = direction(kns)\\
& \wedge\ channel(ev) \subseteq channel(kns)\\
& \wedge\ role(ev) = role(kns)\\
& \wedge\ intent(ev) = intent(kns)\\
& \wedge\ form(ev) = form(kns)\\
& \wedge\ claimSort(ev) = claimSort(kns)\\
& \wedge\ (\forall j : components(kns, j) = H\\
& \quad \Rightarrow components(ev, j) = H)\\
& \wedge\ (\forall i : varnames(extractName(kns, i)) \neq \emptyset\\
& \quad \Rightarrow extractName(ev, i) = extractName(kns, i)).
\end{aligned}$$

The first line states that either the event and the known step both describe incoming events or both describe outgoing events. The next line says that if it is specified in the known step that an event can only happen with respect to one kind of channel, this has to be equal in the event added to the role specification. If *channel* returns several options of what channels can be used in the step, the event's channel, which is always unique, has to match one of them. The third line denotes that the role of the event has to remain the same. Next, it is stated that intention is specified in both the event and the known step or in neither of them. Also, the event has to have the same form as the known step, that is they are both concatenations or both captchas of the same amount of terms. The next line says that the sort of claim of the event and known step have to be equal. That is, either both denote a claim of the same kind or neither of them describes a claim. Further, whenever the term H occurs in the known step, the human knows that his own role is denoted by it. For this reason, a substituted event has to have H at the same positions. Finally, the last lines say that whenever there is a message that is already named with a concrete variable name in the known step, the message at the corresponding position in the role event has to keep the same name.

Example 4.18. Let $kns_1 = In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle\text{'}candidate\text{'},c\rangle, \langle yn,y\rangle\rangle)$ and $kns_2 = Out_H(To(\langle xn,x\rangle), \langle\text{'}code\text{'},code\rangle)$ (as in the countermeasure of Example 4.16).
Let further $ev_1 = In\_S_H(\langle\text{'}D\text{'},D\rangle, \langle\text{'}H\text{'},H\rangle, \langle\langle\text{'}S\text{'},S\rangle, \langle\text{'}candidate\text{'},c\rangle, \langle\text{'}H\text{'},H\rangle\rangle)$ and $ev_2 = Out_H(\langle\text{'}code\text{'},code\rangle)$

We observe that $ev_1 \in accepted(kns_1)$, because:

$direction(ev_1) = direction(kns_1) = in,$
$channel(ev_1) = \{secure\} \subseteq \{secure\} = channel(kns_1),$
$intent(ev_1) = intent(kns_1) = 0,$
$role(ev_1) = role(kns_1) = H,$
$form(ev_1) = form(kns_1) =$
$[(concatenation, 3), (concatenation, 1), (concatenation, 1), (concatenation, 1)],$
$claimSort(ev_1) = claimSort(kns_1) = \{\}$
$components(ev_1, 3) = components(kns_1, 3) = H,$
and $extractName(ev_1, 3) = extractName(kns_1, 3) =$ *'candidate'.*

Because 'candidate' is the only variable name in the known step and because no more occurrences of H are specified, all conditions of Definition 32 are met. In contrast to this, $ev_2 \notin accepted(kns_2)$, because $intent(kns_2) = 1$ while $intent(ev_2) = 0$ which contradicts the fourth condition of the definition, namely that whenever the known step has intention the event also has to have intention.

With this we have a way to state what role events the adversary can add to the human role specification for what known steps in the countermeasure. Additionally, we have to respect the order of the known steps and the noDo part of the countermeasure. We give a full description of what an adversary is allowed to do in the following section.

4.3.2 Derivable Human Role Specifications

For defining the derivable human role specifications, we introduce the function $exception(K(H))$ which given a countermeasure, returns a set of all known steps that describe an exception. Thereby, we consider those known steps to describe an exception which contain a claim or which state an exception to a noDo knowledge rule. That is, all known steps containing a claim are returned, as well as all known send steps containing a message with a name $n \in NaVar$ that is also specified in a $NoTell$ and all known receive steps containing a message name which also occurs in a $NoGet$. We further generalize the function $extractName$ which extracts all the names of terms used in an event, to also take a role specification as input.

We are given a fixed protocol P and the role specification $P(H) = (IK, \rho)$ for the role of the human H, where IK is the initial knowledge and $\rho = [ev_1, ..., ev_n]$ a sequence of events $ev_i \in RoleEvent$ such that (IK, ρ) is well-formed according to Definition 7. We are additionally given a countermeasure $K(H) = (nDK, \sigma)$ where nDK is a set of noDo knowledge rules and $\sigma = [s_1, ..., s_n]$ a sequence of known steps.
The set $HRoles(P, H, K)$ contains all human role specifications that are allowed to be derived from the role specification $P(H)$, conditioned on the countermeasure $K(H)$. In order to define this set, we first define what role specifications (IK, ρ') are derivable by the adversary and can be an element of $HRoles(P, H, K)$.

Definition 33. *Given the role specification (IK, ρ), role specification (IK, ρ') is derivable from it if it is well-formed and can be obtained by*

- *deleting any events $ev_i \in \rho$,*
- *inserting new events $ev \in \mathit{RoleEvent}_H$ to the sequence of events ρ,*
- *and changing the order of the events $ev_i \in \rho$,*

subject to the following five conditions:

Condition 1. *The adversary can only generate messages with variable names $NaVar$ that are already appearing in the original protocol specification.*

$$\begin{aligned}
&\forall\,(IK, \rho') \in \mathit{HRoles}(H, P, K), na, i :\\
&\quad na = extractName((IK, \rho'), i)\\
&\quad\ \Rightarrow \exists R \in dom(P), j : na = extractName(P(R), j)
\end{aligned}$$

Condition 2. *A send containing a message with the name specified in a $NoTell()$ cannot be added to the sequence of events. The only exception is if the event is accepted by a known step occurring in $K(H)$. The same is true for all messages if $NoTell$ is used without any arguments.*

$$\begin{aligned}
&\forall\,(IK, \rho') \in \mathit{HRoles}(H, P, K), xn, x, Hn, \mathit{R2n}, \mathit{R2}, ev, m :\\
&\quad (NoTell(xn) \in nDK \vee NoTell \in nDK) \wedge ev \in \rho' \wedge\\
&\quad ev \in \mathit{Send}_H(\langle \mathit{Hn,H}\rangle, \langle \mathit{R2n,R2}\rangle, m) \wedge m \vdash_H \langle \mathit{xn,x}\rangle\\
&\quad\ \Rightarrow \exists s : s \in \sigma \wedge ev \in accepted(s)
\end{aligned}$$

Condition 3. *A receive containing a message with the name specified in a $NoGet()$ cannot be added to the sequence of events. The only exception is if the event is accepted by a known step occurring in $K(H)$. The same is true for all messages if $NoGet$ is used without any arguments.*

$$\begin{aligned}
&\forall\,(IK, \rho') \in \mathit{HRoles}(H, P, K), xn, x, Hn, \mathit{R2n}, \mathit{R2}, ev, m :\\
&\quad (NoGet(xn) \in nDK \vee NoGet \in nDK) \wedge ev \in \rho' \wedge\\
&\quad ev \in \mathit{Receive}_H(\langle \mathit{R2n,R2}\rangle, \langle \mathit{Hn,H}\rangle, m) \wedge m \vdash_H \langle \mathit{xn,x}\rangle\\
&\quad\ \Rightarrow \exists s : s \in \sigma \wedge ev \in accepted(s)
\end{aligned}$$

Condition 4. *If the human has $NoOverwrite()$ with a message name as argument in his knowledge, the following must hold: whenever a message with the specified name is already in the initial knowledge or in a previous receiving event, the adversary can only add a receive event of a message with the same name if the values of the messages also match. If $NoOverwrite$ is used without* *any arguments, this is true for all messages.*

$$
\begin{aligned}
&\forall\ (IK, \rho') \in HRoles(H,P,K), xn, x, y, Hn, R2n, R2, R3n, R3,\\
&ev, ev_1, ev_2, m, m_1, m_2 :\\
&\quad ((NoOverwrite(xn) \in nDK \lor NoOverwrite \in nDK)\\
&\qquad \land \langle xn, x\rangle \in IK \land ev \in \rho'\\
&\qquad \land ev \in Receive_H(\langle R2n, R2\rangle, \langle Hn, H\rangle, m) \land m \vdash_H \langle xn, y\rangle\\
&\qquad\quad \Rightarrow x = y)\\
&\quad \land ((NoOverwrite(xn) \in nDK \lor NoOverwrite \in nDK)\\
&\qquad \land ev_1, ev_2 \in \rho'\\
&\qquad \land ev_1 \in Receive_H(\langle R2n, R2\rangle, \langle Hn, H\rangle, m_1) \land m_1 \vdash_H \langle xn, x\rangle\\
&\qquad \land ev_2 \in Receive_H(\langle R3n, R3\rangle, \langle Hn, H\rangle, m_2) \land m_2 \vdash_H \langle xn, y\rangle\\
&\qquad\quad \Rightarrow x = y)
\end{aligned}
$$

Condition 5. *An event which is accepted by a known step describing an exception can only be part of the human role specification if for every preceding known step in the countermeasure there is also an accepted preceding event in the role specification. Thereby, the values substituted for placeholder variables that were equal in the known steps, also have to be equal in all events.*

$$
\begin{aligned}
&\forall\ (IK, \rho') \in HRoles(H,P,K), ev_j, s_l, s_k, :\\
&\quad ev_j \in accepted(s_l) \land s_l \in exception(K(H)) \land s_k \in \sigma \land k < l\\
&\quad \Rightarrow \exists ev_i : ev_i \in \rho' \land i < j \land ev_i \in accepted(s_k)\\
&\qquad \land (\forall m, n : components(s_l, m) = components(s_k, n)\\
&\qquad \Rightarrow components(ev_j, m) = components(ev_i, n))
\end{aligned}
$$

With this, we define $HRoles(H,P,K)$ as follows.

Definition 34. *The set $HRoles(H,P,K)$ is the set of all derivable human role specifications up to α-equivalence of variable names.*

This means that two role specifications only differing in the denotation of the variables are defined to be the same.
Given the set of derivable human role specifications, the role substitution attack is defined as follows.

Definition 35. *In a **role substitution attack**, an adversary can choose any human role specification from the set $HRoles(H,P,K)$ and let it be instantiated at runtime.*

Example 4.19. Let us recall the original human role specification as well as a countermeasure for the protocol *hisp_codevoting*.

$$
\begin{aligned}
&\mathit{hisp\_codevoting}(H) = \\
&\quad (\{\}, \\
&\quad [\, \mathit{In\_S}_H(\langle\text{‘D’},D\rangle, \langle\text{‘H’},H\rangle, \langle\langle\text{‘S’},S\rangle, \langle\text{‘candidate’},c\rangle, \langle\text{‘code’},\mathit{code}\rangle\rangle), \\
&\quad \mathit{Out}_H(\mathit{To}(\text{‘S’},S), \langle\text{’code’},\mathit{code}\rangle)\,]\,) \\
&K(H) = \\
&\quad (\{\mathit{NoGet}(\text{‘candidate’}), \mathit{NoTell}(\text{‘candidate’}), \mathit{NoTell}(\text{‘code’}), \\
&\quad \mathit{NoOverwrite}\}, \\
&\quad [\mathit{In\_S}_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn,x\rangle, \langle\text{‘candidate’},c\rangle, \langle yn,y\rangle\rangle), \\
&\quad \mathit{Out}_H(\mathit{To}(\langle xn,x\rangle), \langle\text{’code’},\mathit{code}\rangle)\,]\,)
\end{aligned}
$$

The following role specification is a valid element of the set $HRoles(P, H, K)$ of possible role specifications that an adversary is allowed to derive:

$$
\begin{aligned}
&\mathit{hisp\_codevoting}(H) = \\
&\quad (\{\}, \\
&\quad [\, \mathit{In\_S}_H(\langle\text{‘D’},D\rangle, \langle\text{‘H’},H\rangle, \langle\langle\text{‘S’},S\rangle, \langle\text{‘candidate’},c\rangle, \langle\text{‘code’},\mathit{code}\rangle\rangle), \\
&\quad \mathit{In}_H(\langle\text{‘S’},S\rangle), \\
&\quad \mathit{Out}_H(\mathit{To}(\text{‘S’},S), \langle\text{‘code’},\mathit{code}\rangle), \\
&\quad \mathit{Out}_H(\langle\text{‘S’},S\rangle)\,]\,)
\end{aligned}
$$

The following role specification is **not** a valid element of $HRoles(P, H, K)$:

$$
\begin{aligned}
&\mathit{hisp\_codevoting}(H) = \\
&\quad (\{\}, \\
&\quad [\, \mathit{In\_S}_H(\langle\text{‘D’},D\rangle, \langle\text{‘H’},H\rangle, \langle\langle\text{‘S’},S\rangle, \langle\text{‘candidate’},c\rangle, \langle\text{‘code’},\mathit{code}\rangle\rangle), \\
&\quad \mathit{Out}_H(\langle\text{’candidate’},c\rangle), \\
&\quad \mathit{Out}_H(\langle\text{’code’},\mathit{code}\rangle)\,]\,)
\end{aligned}
$$

Because of the $\mathit{NoTell}(\text{‘candidate’})$, $\langle\text{‘candidate’},c\rangle$ could only be sent if there was a known step describing such an exception, which is not the case. Further, because of the $\mathit{NoTell}(\text{‘code’})$, $\langle\text{‘code’},\mathit{code}\rangle$ can only be sent in an event accepted by the second step in the knowledge. Contradicting this, the event $\mathit{Out}_H(\langle\text{’code’},\mathit{code}\rangle)$ in the role specification is not accepted by this known step because it does not include an intention.

4.3.3 Extra Conditions for a Bounded Verification

Given the conditions defined in the role substitution attack, an adversary can derive an infinite number of possible human role specifications. For this reason, we define in this section parameters which allow to keep the set $HRoles(P, H, K)$ of all possible role specifications bounded. Given this, we can introduce another parameter to ensure bounded verification.
We introduce, in the following, parameters that restrict the number of events allowed in a role specification and the number of function symbols that can be used in a message. We can then give these parameters as an input to the adversary, and only allow him to derive role specifications that fulfill all conditions

from Section 4.3.2 and respect the bounds imposed on these parameters.

Restricting the number of events in a role specification: As a first parameter we introduce the finite number *number_events* which limits the number of events allowed in the role specification.
The number *number_events* thus defines how many steps a human is willing to do in a protocol.

Restricting the number of function symbols: We restrict the length of each event by defining the parameter *number_functionsymbol* which restricts the number of function symbols that can appear in an event. If only a finite number of function symbols is allowed in every event, all of them are bounded. They will always have finite length.

By requiring that every element of $HRoles(P, H, K)$ respects the introduced parameters we make sure that the set of derivable human role specifications is bounded.

Bounded Verification: We can define the finite number $NumRun$ to denote the number of allowed runs during verification. If we also bound the number of derivable human role specifications as just described, this ensures decidable verification because all role specifications are of finite length [16].

5 Tamarin Model

For automatically analyzing the security protocols, we use the rewriting based verification tool *Tamarin* [12]. In order to input the protocols into the tool, we manually translated them from the introduced protocol specifications to the Tamarin specification files [17].
Tamarin uses a trace based system for verification, whose traces can be controlled by a functionality called axioms. In the axioms, logical formulas can be defined to eliminate some of the traces from consideration. We, therefore, implemented the countermeasures by excluding with axioms the traces that do not meet the properties specified by the countermeasures.
We evaluate all protocols under the assumption $Agent_C = \emptyset$. That is, we assume the adversary has not compromised any agents.

In the following section we will give a more detailed description of the Tamarin models for readers who are already familiar with the tool.

5.1 Tamarin Model Details

Agents and Channels: As is standard, we model the non-human roles by agent states that describe the transition of an agent when a particular event from the role specification takes place. To model the secure, authentic and confidential channels, we use rules closely aligned with the channel rules of *Basin et al.* [4].

Human Role: Because we want to examine the protocols with respect to

all possible human role specifications that the adversary is allowed to derive and because the number of these role specifications is arbitrary large, we do not model all of them by agent states. We instead model human knowledge by *!HK($H,$mn,mv)* facts, that store the human role name H, as well as the name mn and the value mv of the stored message. The 'free human rules' then state that at any time new messages sent to the human on any channel can be added to his knowledge, and stored messages can be sent out on any channel. In particular, the adversary can store every message to and extract every message from the human knowledge with these rules. Further, there is a rule allowing the human to produce new fresh values and store them to his knowledge.
In summary, instead of examining all human role specifications the adversary is allowed to derive, the Tamarin models cover every possible human role specification by the free human rules.

Countermeasures: The noDo countermeasures, expressing what a human will never do, are expressed by axioms. For this purpose, the action *H($H)* is added to all free human rules, the action *Comm($H,$mn,mv)* to the rules where the human *$H* sends a message with name mn and value mv and the action *Learn($H,$mn,mv)* to the rules where he receives such a message. Further the action *HKn($H,$mn,mv)* is added to every rule where the message with name mn and value mv is stored newly to the knowledge of the human *$H*. This includes all rules where such a message is received or freshly produced as well as the setup rule, where the initial knowledge is produced. As an example, we present axioms describing *NoTell*, *NoGet* and *NoOverwrite*. Because they can all be used with a concrete variable name as input, we take 'vote' as an example and define *NoTell('vote')* and *NoGet('vote')* with respect to it.

```
axiom NoTellVote:
    "All H x #j.
        Comm(H,'vote',x) @j & H(H) @j ⇒ F"

axiom NoGetVote:
    "All H x #j.
        Learn(H,'vote',x) @j & H(H) @j ⇒ F"

axiom NoOverwrite:
    "All H name x y #i #j.
        HKn(H,name,x) @i & HKn(H,name,y) @j ⇒ x=y"
```

With this, the axioms $NoTell$ and $NoGet$ eliminate all traces, where a message with the specified name (here 'vote') is sent or received in a free human rule. $NoOverwrite$ makes sure that whenever a message is stored in the human knowledge, a message with the same name can only be stored, if it has the same value. This eliminates thus all the traces where a message in the human knowledge is overwritten with a new value.

The known steps in the countermeasures are modeled similarly to agent states. The difference is that only those terms are passed from one state to the next, which are denoted by the same variable in the countermeasure. With this, we ensure that their instantiation remains the same over several known steps. Further, all placeholder variables from the countermeasure are modeled by variables

such that everything can be sent or received for them.
With these rules, we can express the exceptions that even if a message's name is occurring in a $NoTell()$ or $NoGet()$ rule, it can still be sent or received in the context specified by the known steps. These traces are not eliminated, because the action *H($H)* is not included in the known step rules. Further, we capture the requirement of the countermeasure that a known step describing an exception is only performed when the previous ones are also done.

Implementation Details: One thing which is different in the Tamarin implementation compared to the formal model is which known steps are defining an exception to the noDo knowledge rules. We defined in the formal model that the message names which occur in a $NoTell$ or $NoGet$ rule can only be contained in an event of the role specification which is described by a known step that explicitly uses the same variable name. In contrast to this, we saw that in the Tamarin implementation, the names specified in $NoTell$ or $NoGet$ can occur in all known step rules. This is the case because the axioms for $NoTell$ and $NoGet$ only exclude the rules which have an action *H($H)* and an action $Comm()$ or $Learn()$ of the corresponding name. However, the known steps do not have an action *H($H)*. Summing up, this means that in the Tamarin models all known steps can describe an exception for a noDo rule, while only the known steps explicitly mentioning the specified name do so in the formal model. Because it means that more traces are possible in the Tamarin model, we do not consider this to be a problem.

Another way in which the Tamarin implementation differs from the formal model is the realization of the messages' names. While in the formal model a composed term can consist of different sub-terms that all have a name, in the Tamarin implementation every message, including the composed ones, has one single name. This means that all messages are expressed as a pair where the first component denotes a name for the whole composed term and the second component its value. For convenience, we have not adjusted the names in the implementation, because we do not think that this has an impact of what attacks are possible.

Adversary Model: In the Tamarin model, we examined the security protocols with respect to an adversary that controls the network but who does not compromise participating agents. We saw that even with this simplification the role substitution attack breaks many protocols. A next step would be to consider an even more powerful adversary which is able to compromise some participating agents. Our formal model does not have to be changed to model an adversary that can compromise selected agents and our definitions of the security properties already take such a capability into consideration.

6 Robuster Protocols

Instead of defining countermeasures that provide the human with more knowledge, the possibilities of an adversary can sometimes be restricted by changing the original protocol specification. The goal, thereby, is that the non-human roles share enough knowledge about the protocol that producing all the mes-

Figure 5: Protocol *hisp_codevoting_robuster*

0.	D :	knows($\langle H, S, c, h(c)\rangle$)
0.	S :	knows($\langle H, D, \boldsymbol{P}, c, h(c), \boldsymbol{pk(ltkP)}\rangle$)
0.	**P** :	**knows**($\boldsymbol{\langle H,S,ltkP\rangle}$)
1.	D $\bullet\!\!\rightarrow\!\!\bullet$ H :	$\langle S, c, h(c)\rangle / \langle S, c, code\rangle$
2.	**H** $\bullet\!\!\rightarrow\!\!\circ$ **P** :	$code$
3.	P $\circ\!\!\rightarrow\!\!\circ$ S :	$code, \boldsymbol{\{code\}_{sk(P)}} / h(c), \boldsymbol{\{h(c)\}_{sk(P)}}$

sages they expect only allows for a limited adversarial variation of the human role specification. We define the following notion of robustness:

Definition 36. *Protocol P1 is said to be* ***more robust*** *than protocol P2 if in the countermeasure of protocol P1 a subset of the knowledge specified in the countermeasure of protocol P2 is needed to satisfy the same security properties.*

In this section, we introduce ideas how the original protocol specifications could be changed to achieve more robust protocols. We then show an example protocol in which the demonstrated ideas are applied effectively.

Adding cryptographic mechanisms: One possibility to achieve more robust protocols is to add signatures or encryptions to the communication between non-human entities. Adding a signature can help that a non-human receiver knows a message was authentically sent by another non-human agent and cannot come from the adversary. Conversely, a public encryption can help to ensure that only the intended receiver can read the message. As a result it might be possible to avoid attacks where an adversary learns a message to reuse it later.

Strengthen the communication channels: Another measure that can help to make protocols more robust is to strengthen the assumptions on the communication channels. This means we can require channels to be authentic, confidential, or even secure. As a result the agents of a protocol have stronger guarantees of who communicated with them or who can read the messages they sent, which can avoid attacks.

Example 6.1. The protocol *hisp_codevoting* from Example 3.1 can be made more robust by adding a signature to the last message, where the code is sent from platform P to the server S, and by assuming that the channel from the human H to the platform P is authentic. The Alice&Bob notation of the resulting protocol *hisp_codevoting_robuster* is shown in Figure 5, where the new elements are shown in bold.
Under the assumption that the adversary has not compromised any agents, we have shown with the Tamarin tool that without countermeasures the new version of the protocol satisfies the claim *authentic_hisp_withName* of S with respect to H and the code $h(c)$. At the same time, only the corresponding claim *alive* of S is satisfied in the original protocol. The following attack falsifies the claim *authentic_hisp* in the original protocol version: An adversary can learn from the human all candidate-code pairs and then send the code corresponding to his own candidate choice to the platform, which relays it to the server. Consequently, the

server S wrongly thinks that the candidate vote it receives reflects the choice of the human H. This is not possible anymore in *hisp_codevoting_robuster*, because the channel from the human to the platform P is assumed to be authentic. Further, it is also not possible for the adversary to send his own choice directly to the server S, because S only accepts a code together with a matching signature from the platform P.
We suspect that it is not possible to achieve *authentic_hisp_agreement* without a human having in his knowledge at least a known step where he sends the ⟨*'code',code*⟩ intentionally to ⟨*'S',S*⟩ and a $NoTell$ which avoids that he sends the same message in any other context. Therefore, we conjecture that the protocol cannot be made more robust to also satisfy *authentic_hisp_agreement* with no human knowledge, even though we have no formal proof for this fact.

Adding more messages: Another way which can make protocols more robust is to add more messages to the communication between agents. This can help in that two agents might now agree on messages which they could not before because the messages have never been communicated between them.
In Section 8.2.3, we show in the context of the presented case study how such a countermeasure can be effective in making a protocol more robust.

We conclude that as an alternative to the countermeasures that describe the human's knowledge, it is sometimes possible to change the original specifications such that more robust protocols result, where less knowledge of the human is required to achieve the security goals.

7 Model Extension

In this section we extend the introduced model to express more detailed countermeasures. We present means to express that a human knows he will send or receive a message but he does not know whether this message is part of a bigger composed message or not.

Recall that every known step in a countermeasure describes the form of the message that is sent or received by the human. This means that the human always knows what parts of the message consist of concatenations and captchas and further knows how many components the concatenations and captchas have. We now introduce means to express that a human knows he can send or receive a message with a specified name, but without knowing in what form the communication will happen. For this purpose we extend our model as follows.
First, we introduce *include(NaVar)* to be a possible known step of the countermeasures. The argument $NaVar$ specifies the name of a message which is known to be contained in the sent or receive event described by this known step. This name always has to be stated by a fixed variable name of type $NaVar$, rather than by an unknown placeholder variable as is allowed elsewhere in the countermeasures.

Example 7.1. The known step S •→• H: *include('message')* describes that a human knows he can receive from role S a message named *'message'* over a secure channel. He does not know whether he receives the message by itself or in a concatenation or captcha together with other messages. H $\bullet\!\!\rightarrow\!\!\bullet$ S: *include*(*'message'*) denotes the same for a send event.

When expressing the countermeasures formally, we also adopt the function *include* and place it in the last part of the known step, which is describing the communicated message.

Example 7.2. The known steps S $\bullet\!\!\rightarrow\!\!\bullet$ H: *include*(*'message'*) and H $\bullet\!\!\rightarrow\!\!\bullet$ S: *include*(*'message'*) are respectively translated into the formal known steps $In\_S_H(\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, include(\text{'message'}))$ and $Out\_S_H(\langle \text{'H'},H\rangle, \langle \text{'S'},S\rangle,$ $include(\text{'message'}))$.

Because known steps can now contain an *include*, we have to adjust what role events are accepted for a known step. We introduce the function $incl : KnownStep \rightarrow NaVar \cup \{\}$ which given a known step returns the message name specified in the *include* of the known step or the empty set if there is no *include* specified. Recall the function *components* which returns for a known step or a role event a sequence of all its components. We specify that whenever the last argument of a known step is an *include*, no components are extracted by the function *components* for this part of the known step. Additionally, we define *componentsMsg* to do the same as the original *components* but only for the part of the known step or role event which denotes the communicated message, that is for the last argument of the known step or event. We use the function *componentsMsg* with a second argument i to denote the $(i+1)$th element of the sequence.

Example 7.3.

$incl(In\_S_H(\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, include(\text{'message'})) = \text{'message'}$

$components(In\_S_H(\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, include(\text{'message'})))) = [\text{'S'},S,\text{'H'},H]$

$componentsMsg(In\_S_H(\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, captcha(\langle \text{'message1'},m1\rangle,$
$\langle \text{'message2'},m2\rangle))) = [\text{'message1'},m1,\text{'message2'},m2]$

$componentsMsg(In\_S_H(\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, captcha(\langle \text{'message1'},m1\rangle,$
$\langle \text{'message2'},m2\rangle)), 2) = \text{'message2'}$

The definition of accepted is then changed as follows.

Definition 37. *We say that an event $ev \in RoleEvent$ is* ***accepted*** *for a known step $kns \in KnownStep$ if the following is given:*

$$\begin{aligned}
ev \in accepted(kns) \Longleftrightarrow\; & direction(ev) = direction(kns)\\
& \wedge\; channel(ev) \subseteq channel(kns)\\
& \wedge\; role(ev) = role(kns)\\
& \wedge\; intent(ev) = intent(kns)\\
& \wedge\; claimSort(ev) = claimSort(kns)\\
& \wedge\; (\forall j : components(kns, j) = H\\
& \quad \Rightarrow components(ev, j) = H)\\
& \wedge ((incl(kns) = \emptyset \wedge form(ev) = form(kns)\\
& \quad \wedge (\forall i : varnames(extractName(kns, i)) \neq \emptyset\\
& \qquad \Rightarrow extractName(ev, i) = extractName(kns, i)))\\
& \quad \vee \exists i : incl(kns) = componentsMsg(ev, i))
\end{aligned}$$

The first part of the definition remains unchanged and describes that both the known step *kns* and the event *ev* have to be incoming or outgoing, that the channel of *ev* has to equal an allowed channel of *kns*, that the role of *kns* and *ev* have to be the same, that either an intention is specified in both *kns* and *ev* or in none of them, and that either both *ev* and *kns* describe a claim of the same kind or none of them describes a claim. Also the next line is as in the original definition and says that every occurrence of H has to remain in the substituted event. The only difference is that the definition of *components* has been adjusted to only consider the first parts of a known step if an *include* is contained in it. The last part states that if no *include* is specified in the known step, the form of *kns* and *ev* have to match and every fixed variable name has to be unchanged in the event, as before. Alternatively, the known step can now contain an *include* and the specified message has to be contained in the message part of the event.

Example 7.4. Let $kns_1 = In\_S_H(\langle Sn,S\rangle, \langle Hn,H\rangle, include(\text{`}message1\text{'}))$, $ev_1 = In\_S_H(\langle \text{`}S\text{'},S\rangle, \langle \text{`}H\text{'},H\rangle, captcha(\langle \text{`}message1\text{'},m1\rangle, \langle \text{`}message2\text{'},m2\rangle)$ and $ev_2 = In\_S_H(\langle \text{`}S\text{'},S\rangle, \langle \text{`}H\text{'},H\rangle, captcha(\langle \text{`}message2\text{'},m2\rangle)$.

Then, $ev_1 \in accepted(kns_1)$, because

$$\begin{aligned}
&direction(ev_1) = direction(kns_1) = in,\\
&channel(ev_1) = \{secure\} \subseteq \{secure\} = channel(kns_1),\\
&role(ev_1) = role(kns_1) = H,\\
&intent(ev_1) = intent(kns_1) = 0,\\
&components(kns_1, 3) = components(ev_1, 3) = H
\end{aligned}$$

and there are not more occurrences of the role H, and

$$incl(kns_1) = componentsMsg(ev_1, 0) = \text{`}message1\text{'}.$$

At the same time $ev_2 \notin accepted(kns_1)$ because $incl(kns_1) = \text{`}message1\text{'}$ but there exist no $i$ such that $componentsMsg(ev_2, i) = \text{`}message1\text{'}$ is given.

With this definition of *accepted*, the role substitution attack can be defined as before and the rest of the model remains unchanged. We will use the model extension in the following case study to express that a human knows he will receive a certain message on his platform before performing a claim, but he does not know in what form he will receive it.

8 Case Studies

In this section, we present two case studies in which we examine an e-voting and an e-banking protocol with our model. We analyze the protocols with the Tamarin tool and assume that there are no compromised agents participating in the protocol, that is $Agent_C = \emptyset$. We, thus, show how the role substitution attack can be used to model specific human errors and we illustrate a strategy to make protocols more robust.

Figure 6: Protocol *evoting_symbol*

0.	H :	knows($\langle D, P, S, vote, birthday \rangle$)
0.	D :	knows($\langle H, ID, symbol, code, barcode \rangle$)
0.	S :	knows($\langle H, D, ID, symbol, birthday, code, barcode \rangle$)
1.	D $\bullet\!\!\rightarrow\!\!\bullet$ H :	$\langle ID, symbol, code, barcode \rangle$
2.	H $\bullet\!\!\rightarrow\!\!\bullet$ P :	$\langle S, ID, vote \rangle$
4.	P $\bullet\!\!\rightarrow\!\!\bullet$ S :	$\langle ID, vote \rangle$
5.	S $\bullet\!\!\rightarrow\!\!\bullet$ P :	$captcha(vote, symbol)/captcha$
6.	P $\bullet\!\!\rightarrow\!\!\bullet$ H :	$captcha/captcha(vote, symbol)$
7.	H $\bullet\!\!\rightarrow\!\!\bullet$ P :	$\langle birthday, code \rangle$
8.	P $\bullet\!\!\rightarrow\!\!\bullet$ S :	$\langle birthday, code \rangle$
9.	S $\bullet\!\!\rightarrow\!\!\bullet$ P :	$\langle barcode, symbol \rangle$
10.	P $\bullet\!\!\rightarrow\!\!\bullet$ H :	$\langle barcode, symbol \rangle$

8.1 *Evoting_symbol*

The first protocol we examine is a model of the precursor of a deployed e-voting system [2]. The protocol *evoting_symbol* describes an e-voting protocol in which a human H wants to communicate his vote to a server S. H has secure access to a device D, for example in the form of a letter, which provides him with information. Further, H has access to a platform P through which he can communicate with the server. We assume secure channels between H and D, between H and P, and between P and S.
We first introduce the protocol and show the role specifications. Then, we analyze the protocol first under the assumption that the human has no knowledge and then with respect to a countermeasure. Next, we examine some variations including a version where we model by the countermeasures the given instructions in the realization of the protocol. Finally, we show how to model one particular error of the human with our model and examine a protocol where the human accidentally enters a wrong server name (e.g. by mistyping the server's URL).

8.1.1 Protocol Specification

Figure 6 depicts the Alice&Bob notation of *evoting_symbol.* First, the human H reads from the device D his ID, a *symbol*, a *code* and the numerical representation of a *barcode*. Then, he sends his vote, together with S and his ID to the platform P. P transfers the ID and the *vote* to the server S. Upon receiving this message, S forms a captcha of the *vote* and the *symbol* and sends it back to H, via P. Following this, H enters *birthday* and *code* which is again sent to S through the platform P. As a final step, S then sends the *barcode* and *symbol* back to H.

In the realization of this protocol, the *symbol* is supposed to be an image that is easy recognizable and comparable for the human even if its shape is warped or stretched in a captcha. At the same time it is too detailed to be explained to somebody who has not seen it. This should prevent that an adversary can

learn a *symbol* by asking it from a human to then use it in an attack.
For modeling this idea, we add a *NoTell('symbol')* as a ground assumption to the protocol. Further, we also assume *NoOverwrite*, that is we say that a human will not change the values of messages stored in his knowledge. This ensures that he will for example not read the information in the first step from two different devices.

The following are the role specifications for all participating roles:

$evoting\_symbol(D) =$
$(\{\langle\langle 'H',H\rangle, \langle 'ID',ID\rangle, \langle 'symbol',symbol\rangle, \langle 'code',code\rangle, \langle 'barcode',barcode\rangle\rangle\},$
$[\ Out\_S_D(\langle 'D',D\rangle, \langle 'H',H\rangle, \langle\langle 'ID',ID\rangle, \langle 'symbol',symbol\rangle, \langle 'code',code\rangle,$
$\langle 'barcode',barcode\rangle\rangle)\ ]\ )$

$evoting\_symbol(P) =$
$(\{\},$
$[\ In\_S_P(\langle 'H',H\rangle, \langle 'P',P\rangle, \langle\langle 'S',S\rangle, \langle 'ID',ID\rangle, \langle 'vote',vote\rangle\rangle),$
$Out\_S_P(\langle 'P',P\rangle, \langle 'S',S\rangle, \langle\langle 'ID',ID\rangle, \langle 'vote',vote\rangle\rangle),$
$In\_S_P(\langle 'S',S\rangle, \langle 'P',P\rangle, captcha),$
$Out\_S_P(\langle 'P',P\rangle, \langle 'H',H\rangle, captcha),$
$In\_S_P(\langle 'H',H\rangle, \langle 'P',P\rangle, \langle\langle 'birthday',birthday\rangle, \langle 'code',code\rangle\rangle),$
$Out\_S_P(\langle 'P',P\rangle, \langle 'S',S\rangle, \langle\langle 'birthday',birthday\rangle, \langle 'code',code\rangle\rangle),$
$In\_S_P(\langle 'S',S\rangle, \langle 'P',P\rangle, \langle\langle 'barcode',barcode\rangle, \langle 'symbol',symbol\rangle\rangle),$
$Out\_S_P(\langle 'P',P\rangle, \langle 'H',H\rangle, \langle\langle 'barcode',barcode\rangle, \langle 'symbol',symbol\rangle\rangle)\ ]\ )$

$evoting\_symbol(S) =$
$(\{\langle\langle 'H',H\rangle, \langle 'D',D\rangle, \langle 'ID',ID\rangle, \langle 'symbol',symbol\rangle,$
$\langle 'birthday',birthday\rangle, \langle 'code',code\rangle, \langle 'barcode',barcode\rangle\rangle\},$
$[\ In\_S_S(From(\langle 'H',H\rangle), \langle 'P',P\rangle, \langle 'S',S\rangle, \langle\langle 'ID',ID\rangle, \langle 'vote',vote\rangle\rangle),$
$Out\_S_S(To(\langle 'H',H\rangle), \langle 'S',S\rangle, \langle 'P',P\rangle,$
$captcha(\langle 'vote',vote\rangle, \langle 'symbol',symbol\rangle)),$
$In\_S_S(\langle 'P',P\rangle, \langle 'S',S\rangle, \langle\langle 'birthday',birthday\rangle, \langle 'code',code\rangle\rangle),$
$Claim_S(running, \langle 'S',S\rangle, \langle 'H',H\rangle, \langle 'vote',vote\rangle),$
$Claim_S(authentic\_hisp\_agreement, \langle 'H',H\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle),$
$Out\_S_S(\langle 'S',S\rangle, \langle 'P',P\rangle, \langle\langle 'barcode',barcode\rangle, \langle 'symbol',symbol\rangle\rangle),$
$Claim_S(secret, \langle 'vote',vote\rangle)\ ]\ )$

$evoting\_symbol(H) =$
$(\{\langle\langle\text{'D'},D\rangle, \langle\text{'P'},P\rangle, \langle\text{'S'},S\rangle, \langle\text{'vote'},vote\rangle, \langle\text{'birthday'},birthday\rangle\rangle\},$
$[\ In\_S_H(\langle\text{'D'},D\rangle, \langle\text{'H'},H\rangle, \langle\langle\text{'ID'},ID\rangle, \langle\text{'symbol'},symbol\rangle, \langle\text{'code'},code\rangle,$
$\langle\text{'barcode'},barcode\rangle\rangle),$
$Out\_S_H(To(\langle\text{'S'},S\rangle), \langle\text{'H'},H\rangle, \langle\text{'P'},P\rangle, \langle\langle\text{'S'},S\rangle, \langle\text{'ID'},ID\rangle, \langle\text{'vote'},vote\rangle\rangle),$
$In\_S_H(From(\langle\text{'S'},S\rangle), \langle\text{'P'},P\rangle, \langle\text{'H'},H\rangle,$
$captcha(\langle\text{'vote'},vote\rangle, \langle\text{'symbol'},symbol\rangle)),$
$Out\_S_H(\langle\text{'H'},H\rangle, \langle\text{'P'},P\rangle, \langle\langle\text{'birthday'},birthday\rangle, \langle\text{'code'},code\rangle\rangle),$
$In\_S_H(\langle\text{'P'},P\rangle, \langle\text{'H'},H\rangle, \langle\langle\text{'barcode'},barcode\rangle, \langle\text{'symbol'},symbol\rangle\rangle),$
$Claim_H(secret, \langle\text{'vote'},vote\rangle),$
$Claim_H(authentic\_hisp\_agreement, \langle\text{'S'},S\rangle, \langle\text{'H'},H\rangle, \langle\text{'vote'},vote\rangle),$
$Claim_H(agreement\_message, \langle\text{'S'},S\rangle, \langle\text{'H'},H\rangle, \langle\text{'vote'},vote\rangle)\ ]\ )$

As captured by the role specifications, the goals of the protocol are that the vote is kept secret from the perspective of H and S. Further, there should be an *authentic_hisp_agreement* of the *vote* between H and S, which is claimed in both entities. That is, S wants to be sure that H really sent the *vote* intentionally to him and vice versa. For this reason, in H and S the appropriate intentions are added to the events that send and receive $\langle\text{'vote'},vote\rangle$. Finally, H also claims *agreement_message*, that is H wants to be sure that S accepted the vote he has previously sent.

8.1.2 Analysis

First, we analyzed the protocol with Tamarin with respect to the original human role specification only. Next, we examined it with respect to all possible human role specifications but without any countermeasures, so with the assumption that the human has no knowledge. Apart from the claimed security properties, *secret* and *authentic_hisp_agreement*, we also analyzed the protocol with respect to the weaker authentication properties *alive*, *authentic_hisp* and *authentic_hisp_withName*, from the perspective of both H and S.
Then, we repeated the analysis after having added the countermeasure captured in Figure 7. We denote the new protocol which takes this countermeasure into account *evoting_symbol_fix*. The countermeasure describes that a human knows that he can only send or receive *'vote'* with the exceptions described by the second and third step in his knowledge. The known steps describe that he will first get a concatenation of four messages over a secure channel. If this has happened, he can send the vote together with the intended receiver $\langle xn,x\rangle$ as the first message and another message as the middle message to a secure channel. After that, he can receive a captcha whose intended sender is the same role as the intended receiver in the preceding step and whose first argument is the same vote that he has sent. Finally, the human knows he will receive a pair with the second component being the message $\langle\text{'symbol'},symbol\rangle$ and after that he can perform the claims *authentic_hisp_agreement* and *agreement_message* with respect to *'vote'* and the entity to whom he declared an intention in the second and third step. The star * after $NoTell(\text{'symbol'})$ and $NoOverwrite$ indicates that these are the ground assumptions we made, so they are not newly

Figure 7: Countermeasure in Protocol *evoting_symbol_fix*

$NoTell(\text{‘symbol’})^*$		
$NoOverwrite^*$		
$NotTell(\text{‘vote’})$		
$NotGet(\text{‘vote’})$		
R1 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle xn1,x1\rangle, \langle xn2,x2\rangle, \langle xn3,x3\rangle, \langle xn4,x4\rangle\rangle$	
H $\bullet\!\!\rightarrow\!\!\bullet$ R2:	$\langle\langle xn,x\rangle, \langle yn,y\rangle, \langle \text{‘vote’},vote\rangle\rangle$	$to : \langle xn,x\rangle$
R3 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$captcha(\langle \text{‘vote’},vote\rangle, \langle zn2,z2\rangle)$	$from : \langle xn,x\rangle$
R4 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle un1,u1\rangle, \langle \text{‘symbol’},symbol\rangle\rangle$	
Claim	*(authentic_hisp_agreement,x,H,‘vote’,vote)*	
	(agreement_message,x,H,‘vote’,vote)	

Table 1: Overview Findings Protocol *evoting_symbol*

	evoting_symbol		*evoting_symbol_fix*
	orig. spec.	all spec.	
Claims S			
secret	✓	$\times^{*(1)}$	✓
alive	(✓)	✓	(✓)
authentic_hisp	(✓)	$\times^{*(2)}$	(✓)
authentic_hisp_withName	(✓)	(×)	(✓)
authentic_hisp_agreement	✓	(×)	✓
Claims H			
secret	✓	$\times^{*(1)}$	✓
alive	✓	$\times^{*(4)}$	(✓)
authentic_hisp	(✓)	(×)	(✓)
authentic_hisp_withName	(✓)	(×)	(✓)
authentic_hisp_agreement	✓	(×)	✓
agreement_message	✓	$\times^{*(5)}$	✓

added in the countermeasure, but are there for completeness. Formally, the countermeasure looks as follows:

$$
\begin{aligned}
&K(H) = \\
&\quad (\{NoTell(\text{‘symbol’}), NoOverwrite, NoTell(\text{‘vote’}), NoGet(\text{‘vote’})\},\\
&\quad [\ In\_S_H(\langle R1n,R1\rangle, \langle Hn,H\rangle, \langle\langle xn1,x1\rangle, \langle xn2,x2\rangle, \langle xn3,x3\rangle, \langle xn4,x4\rangle\rangle),\\
&\quad Out\_S_H(To(\langle xn,x\rangle), \langle Hn,H\rangle, \langle R2n,R2\rangle\langle\langle xn,x\rangle, \langle yn,y\rangle, \langle \text{‘vote’},vote\rangle\rangle),\\
&\quad In\_S_H(From(\langle xn,x\rangle), \langle R3n,R3\rangle, \langle Hn,H\rangle, captcha(\langle \text{‘vote’},vote\rangle, \langle zn2,z2\rangle)),\\
&\quad In\_S_H(\langle R4n,R4\rangle, \langle Hn,H\rangle, \langle\langle un1,u1\rangle, \langle \text{‘symbol’},symbol\rangle\rangle),\\
&\quad Claim_H(authentic\_hisp\_agreement, \langle xn,x\rangle, \langle Hn,H\rangle, \langle \text{‘vote’},vote\rangle),\\
&\quad Claim_H(agreement\_message, \langle xn,x\rangle, \langle Hn,H\rangle, \langle \text{‘vote’},vote\rangle)\ ]\ ).
\end{aligned}
$$

Table 1 shows an overview of the results. The checkmark ✓ denotes that a property was verified, and a cross × indicates that an attack has been found.

The checkmarks in parentheses signify that no proof has been done but the property is known to be given because it is implied by a stronger property. Similarly, a cross in parentheses means that an attack to a weaker property exists.
With the original human role specification only ('orig. spec.' in the table), Tamarin finds proofs that all properties in *evoting_symbol* are satisfied. When all human role specifications are allowed ('all spec.'), only *alive* from the perspective of role S is given. For all other claims, attacks can be found which we present in the following.
If the adversary substitutes the human role specification with the following one, the secrecy claims of both H and S (indicated by (1) in the table) are not satisfied, because the human leaks the vote to the adversary.

$$\begin{aligned}
&\mathit{evoting\_symbol}(H) = \\
&\quad (\{\langle\langle \text{'}D\text{'},D\rangle, \langle \text{'}P\text{'},P\rangle, \langle \text{'}S\text{'},S\rangle, \langle \text{'}\mathit{vote}\text{'},\mathit{vote}\rangle, \langle \text{'}\mathit{birthday}\text{'},\mathit{birthday}\rangle\rangle\}, \\
&\quad [\ \mathit{Out}_H(\langle \text{'}\mathit{vote}\text{'},\mathit{vote}\rangle)\ ]\ )
\end{aligned}$$

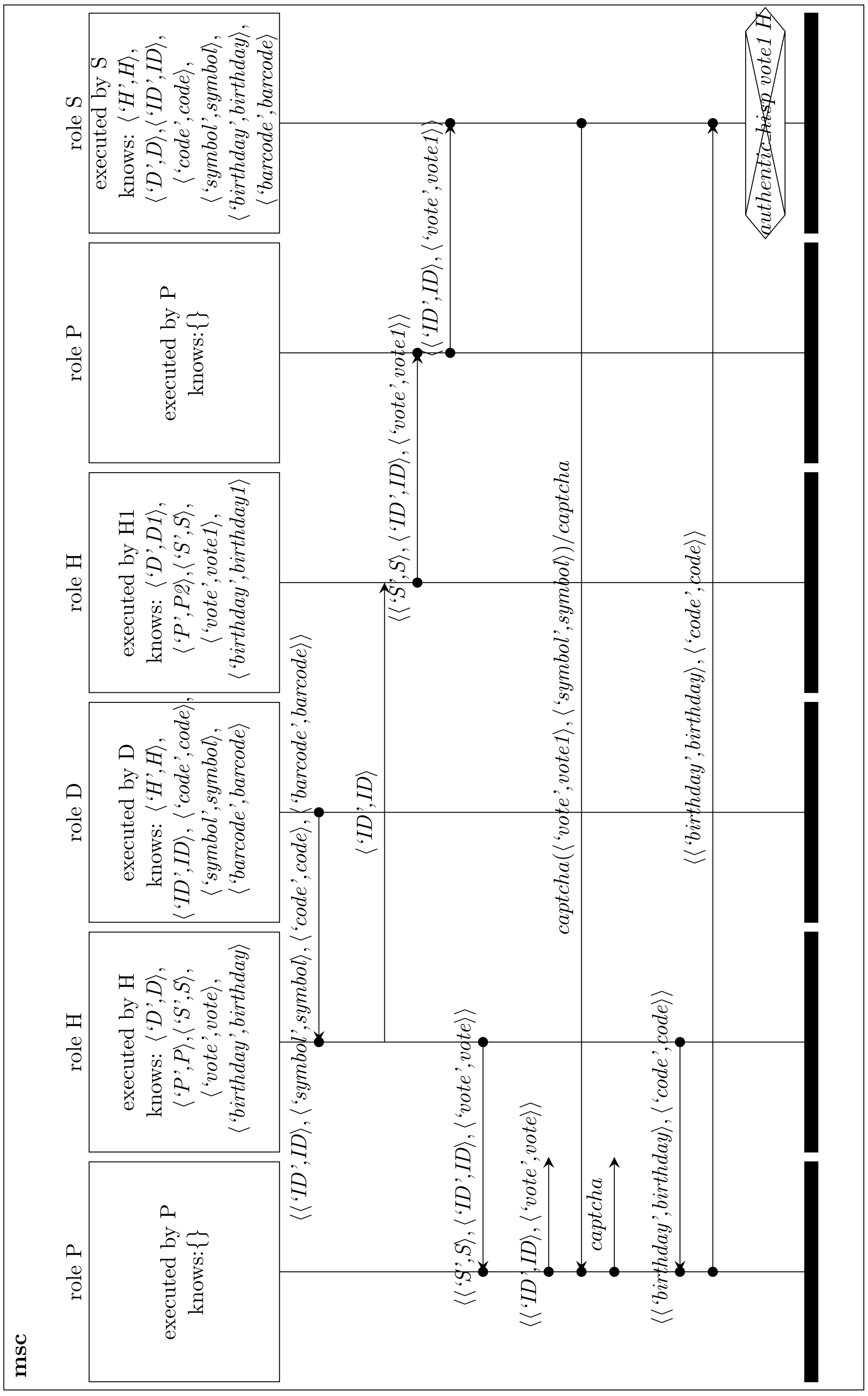

msc
role P
executed by P
knows:{}
role H
executed by H
knows: ⟨'D',D⟩,⟨'P',P⟩,⟨'S',S⟩,⟨'vote',vote⟩,⟨'birthday',birthday⟩
role D
executed by D
knows: ⟨'H',H⟩,⟨'ID',ID⟩,⟨'code',code⟩,⟨'symbol',symbol⟩,⟨'barcode',barcode⟩
role H
executed by H1
knows: ⟨'D',D1⟩,⟨'P',P2⟩,⟨'S',S⟩,⟨'vote',vote1⟩,⟨'birthday',birthday1⟩
role P
executed by P
knows:{}
role S
executed by S
knows: ⟨'H',H⟩,⟨'D',D⟩,⟨'ID',ID⟩,⟨'code',code⟩,⟨'symbol',symbol⟩,⟨'birthday',birthday⟩,⟨'barcode',barcode⟩
⟨⟨'ID',ID⟩,⟨'symbol',symbol⟩,⟨'code',code⟩,⟨'barcode',barcode⟩⟩
⟨'ID',ID⟩
⟨⟨'S',S⟩,⟨'ID',ID⟩,⟨'vote',vote⟩⟩
⟨⟨'S',S⟩,⟨'ID',ID⟩,⟨'vote',vote1⟩⟩
⟨⟨'ID',ID⟩,⟨'vote',vote1⟩⟩
⟨⟨'ID',ID⟩,⟨'vote',vote⟩⟩
captcha(⟨'vote',vote1⟩,⟨'symbol',symbol⟩)/captcha
captcha
⟨⟨'birthday',birthday⟩,⟨'code',code⟩⟩
⟨⟨'birthday',birthday⟩,⟨'code',code⟩⟩
authentic_hisp vote1 H

Figure 8: Protocol *evoting_symbol*: attack on the claim $Claim_S(authentic\_hisp, \langle 'H',H\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle)$ of S

Figure 8 shows an attack on the *authentic_hisp* claim of S ((2) in the table). As in the Alice&Bob notation, the exclusive accesses to channels are denoted by a solid dot. While the claimed human H never sends the vote *vote1* which is received by S, another human *H1* sends it. In order to convince S that the right human H sent the vote, the role specification of H is changed such that he first reveals his ID to another human. Following this, both *H1* and H start a session with platform P. First, *H1* sends the vote *vote1* which is forwarded from P to S that believes it is coming from H. Then another run of the same platform P starts a new session with the human H. The messages sent from P can be used by the adversary to convince S that the previously received *vote1* is coming from the human H. A scenario like this could for example happen if several humans share the same platform. Note that the platform will not recognize the contents of a captcha, so it will not notice that it forwards another vote than it has just sent out. For this attack the substituted role specifications for H are

$$\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle 'D',D\rangle, \langle 'P',P\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle, \langle 'birthday',birthday\rangle\rangle\}, \\
&\quad [\ In\_S_H(\langle 'D',D\rangle, \langle 'H',H\rangle, \langle\langle 'ID',ID\rangle, \langle 'symbol',symbol\rangle, \langle 'code',code\rangle, \\
&\quad \langle 'barcode',barcode\rangle\rangle), \\
&\quad Out_H(\langle 'ID',ID\rangle), \\
&\quad Out\_S_H(\langle 'H',H\rangle, \langle 'P',P\rangle, \langle\langle 'S',S\rangle, \langle 'ID',ID\rangle, \langle 'vote',vote\rangle\rangle), \\
&\quad Out\_S_H(\langle 'H',H\rangle, \langle 'P',P\rangle, \langle\langle 'birthday',birthday\rangle, \langle 'code',code\rangle\rangle)\ ]\ )
\end{aligned}$$

which is instantiated with H and

$$\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle 'D',D\rangle, \langle 'P',P\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle, \langle 'birthday',birthday\rangle\rangle\}, \\
&\quad [\ In_H(\langle 'ID',ID\rangle) \\
&\quad Out\_S_H(\langle 'H',H\rangle, \langle 'P',P\rangle, \langle\langle 'S',S\rangle, \langle 'ID',ID\rangle, \langle 'vote',vote\rangle\rangle)\ ]\ )
\end{aligned}$$

which is instantiated with *H1*.
Finally, the following role specifications contradict the authentication properties *alive* (4) and *agreement_message* (5) from the perspective of role H, because the corresponding claims are done right at the start when no properties are satisfied. This means in particular that no other role was even alive.

$$\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle 'D',D\rangle, \langle 'P',P\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle, \langle 'birthday',birthday\rangle\rangle\}, \\
&\quad [\ Claim_H(alive, \langle 'S',S\rangle, \langle 'H',H\rangle, \langle 'vote',vote\rangle)\ ]\ )
\end{aligned}$$

$$\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle 'D',D\rangle, \langle 'P',P\rangle, \langle 'S',S\rangle, \langle 'vote',vote\rangle, \langle 'birthday',birthday\rangle\rangle\}, \\
&\quad [\ Claim_H(agreement\_message, \langle 'S',S\rangle, \langle 'H',H\rangle, \langle 'vote',vote\rangle)\ ]\ )
\end{aligned}$$

As the name suggests, all evaluated security properties were shown to be satisfied with the countermeasure in the protocol *evoting_symbol_fix*.

8.1.3 Variation: Explainable Symbol

Recall that one of the ground assumptions of the protocol was, that the *symbol* is not explainable to anybody. To find out if this idea helps to avoid attacks

which would otherwise be possible, we examined the protocols from Table 1 again, without including this assumption. The protocols without the assumption are named by appending the suffix *_explainable* to the respective protocol names.
Because without the *NoTell('symbol')*, neither attacks nor proofs could be found with Tamarin for the claims in the protocol *evoting_symbol_fix_explainable*, we made a version of the protocol, where the *NoTell('symbol')* is still included but the *symbol* is leaked from the device to the adversary in the beginning. The idea is that if the adversary can learn the *symbol* directly from the device and this does not help him to produce new attacks, it will also not help him if he can learn the *symbol* from a human.
The analysis showed that it makes no difference for the protocols *evoting_symbol* and *evoting_symbol_fix* whether or not the symbol is explainable when considering the same claims as before. All attacks found in the previous subsection are still possible and all proofs are still verified. However, this does not exclude the possibility that there are other countermeasures for which it would make a difference whether the *symbol* is explainable.

8.1.4 Voting instructions

In the voting instructions of the deployed protocol, the following explanation of the *symbol* was given to the human: *"Bei der Stimmabgabe über Internet erscheint dieses Symbol auf der Bestätigungsseite"*, translated, "when votes are cast over the Internet, this symbol appears on the verification page."
To test whether such an instruction would be sufficient knowledge for the human, we analyzed the protocol with respect to the countermeasure shown in Figure 9. We call the protocol which takes this countermeasure into account *evoting_symbol_realInstruction*. We assume that the human H knows the server S and claims authentication with respect to this role. The claim is only done if the human has received the *'symbol'* in any form from the platform P because he knows from the instruction that he will receive the symbol on a 'verification site'. Further, we assume that the human knows that he should only accept the voting information from the device D, which has to happen before the other known steps. Formally the countermeasure is expressed by:

$$
\begin{aligned}
&K(H) = \\
&\quad (\{NoTell(\textit{`symbol'}), NoOverwrite\}, \\
&\quad [\ In\_S_H(\langle \mathit{Dn,D}\rangle, \langle \mathit{Hn,H}\rangle, \langle\langle \textit{`ID'},\mathit{ID}\rangle, \langle \textit{`symbol'},\mathit{symbol}\rangle, \langle \textit{`code'},\mathit{code}\rangle, \\
&\quad \langle \textit{`barcode'},\mathit{barcode}\rangle\rangle), \\
&\quad In\_Any_H(\langle \mathit{Pn,P}\rangle, \langle \mathit{Hn,H}\rangle, include(\textit{`symbol'})), \\
&\quad Claim_H(authentic\_hisp\_agreement, \langle \mathit{Sn,S}\rangle, \langle \mathit{Hn,H}\rangle, \langle \textit{`vote'},\mathit{vote}\rangle), \\
&\quad Claim_H(agreement\_message, \langle \mathit{Sn,S}\rangle, \langle \mathit{Hn,H}\rangle, \langle \textit{`vote'},\mathit{vote}\rangle)\ ]\ ).
\end{aligned}
$$

Finally, we assume $P \neq D$, which means that the human knows that the device D does not equal the platform P. This helps Tamarin to find more meaningful attacks.

An overview of the findings is shown in Table 2. The only claims that are satisfied are *alive* from the perspective of both H and S. The secrecy claims of

Figure 9: Countermeasure in Protocol *evoting_symbol_realInstruction*

$NoTell($*'symbol'*$)^*$
$NoOverwrite^*$

D •→• H:	⟨⟨*'ID'*,*ID*⟩, ⟨*'symbol'*,*symbol*⟩, ⟨*'code'*,*code*⟩, ⟨*'barcode'*,*barcode*⟩⟩
P ?→? H:	*include('symbol')*
Claim	*(authentic_hisp_agreement,S,H,'vote',vote)*
	(agreement_message,S,H,'vote',vote)

Table 2: Overview Findings Protocol *evoting_symbol_realInstruction*

	evoting_symbol_realInstruction
Claims S	
secret	$\times^{*(1)}$
alive	✓
authentic_hisp	$\times^{*(2)}$
authentic_hisp_withName	(×)
authentic_hisp_agreement	(×)
Claims H	
secret	$\times^{*(1)}$
alive	✓
authentic_hisp	$\times^{*(4)}$
authentic_hisp_withName	(×)
authentic_hisp_agreement	(×)
agreement_message (vote)	$\times^{*(5)}$

both roles (1) are again falsified by the role specification where H just sends the *vote* to the network. The attack to the claim *authentic_hisp* of S (2) is also still the same as shown in the protocol without countermeasures. Because it is not excluded that the human never reaches the known steps in the countermeasure, the role specifications of the previous role substitution attack are still valid with this countermeasure.
An attack to the claim *authentic_hisp* of H (4) is realized by the following substituted human role specification:

$$
\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle \text{'}D\text{'},D\rangle, \langle \text{'}P\text{'},P\rangle, \langle \text{'}S\text{'},S\rangle, \langle \text{'}vote\text{'},vote\rangle, \langle \text{'}birthday\text{'},birthday\rangle\rangle\}, \\
&\quad [\ In\_S_H(\langle \text{'}D\text{'},D\rangle, \langle \text{'}H\text{'},H\rangle, \langle\langle \text{'}ID\text{'},ID\rangle, \langle \text{'}symbol\text{'},symbol\rangle, \langle \text{'}code\text{'},code\rangle, \\
&\quad \langle \text{'}barcode\text{'},barcode\rangle\rangle), \\
&\quad Out\_S_H(\langle \text{'}H\text{'},H\rangle, \langle \text{'}P\text{'},P\rangle, \langle\langle \text{'}S\text{'},S\rangle, \langle \text{'}ID\text{'},ID\rangle, \langle \text{'}vote\text{'},vote\rangle\rangle), \\
&\quad Out\_S_H(\langle \text{'}H\text{'},H\rangle, \langle \text{'}P\text{'},P\rangle, \langle\langle \text{'}birthday\text{'},birthday\rangle, \langle \text{'}code\text{'},code\rangle\rangle), \\
&\quad In\_S_H(\langle \text{'}P\text{'},P\rangle, \langle \text{'}H\text{'},H\rangle, \langle\langle \text{'}barcode\text{'},barcode\rangle, \langle \text{'}symbol\text{'},symbol\rangle\rangle), \\
&\quad Claim_H(authentic\_hisp, \langle \text{'}S\text{'},S\rangle, \langle \text{'}H\text{'},H\rangle, \langle \text{'}vote\text{'},vote\rangle)\ ]\ ).
\end{aligned}
$$

The difference to the original human role specification is that H never receives the captcha of the vote and the symbol. Because H consequently never receives *vote* from the agent in role S, even though S could have sent it, the claim *authentic_hisp* is not satisfied.
Finally, H's claim that he agrees with S on the *vote* is not given (5). A role specification which leads to an attack on this property is the following:

$$
\begin{aligned}
&evoting\_symbol(H) = \\
&\quad (\{\langle\langle \text{'}D\text{'},D\rangle, \langle \text{'}P\text{'},P\rangle, \langle \text{'}S\text{'},S\rangle, \langle \text{'}vote\text{'},vote\rangle, \langle \text{'}birthday\text{'},birthday\rangle\rangle\}, \\
&\quad [\ In\_S_H(\langle \text{'}D\text{'},D\rangle, \langle \text{'}H\text{'},H\rangle, \langle\langle \text{'}ID\text{'},ID\rangle, \langle \text{'}symbol\text{'},symbol\rangle, \langle \text{'}code\text{'},code\rangle, \\
&\quad \langle \text{'}barcode\text{'},barcode\rangle\rangle), \\
&\quad Out\_S_H(\langle \text{'}H\text{'},H\rangle, \langle \text{'}P\text{'},P\rangle, \langle\langle \text{'}S\text{'},S\rangle, \langle \text{'}ID\text{'},ID\rangle, \langle \text{'}vote\text{'},vote\rangle\rangle), \\
&\quad In\_S_H(From(\langle \text{'}S\text{'},S\rangle), \langle \text{'}P\text{'},P\rangle, \langle \text{'}H\text{'},H\rangle, \\
&\quad captcha(\langle \text{'}vote\text{'},vote\rangle, \langle \text{'}symbol\text{'},symbol\rangle)), \\
&\quad Claim_H(agreement\_message, \langle \text{'}S\text{'},S\rangle, \langle \text{'}H\text{'},H\rangle, \langle \text{'}vote\text{'},vote\rangle)\ ]\ )
\end{aligned}
$$

H knows that he can only do the claim after receiving *symbol* from P. Because this applies after H gets a captcha from P which includes *symbol*, H can do an *agreement_message* claim right after this event. At this moment, however, S has not yet performed a running claim and thus not yet counted the *vote*, which it only does after receiving *birthday* and *code* back from H. This means that after seeing *symbol*, the human wrongly concludes that the protocol has finished and that his *vote* was taken into account by S.

8.1.5 One Specific Human Error: Mistyping

In this section we want to show how we can use the role substitution attack to model one particular error of the human instead of considering all possible human role specifications. That is, we present one particular role specification the adversary is allowed to produce in a role substitution attack, which models that the human knows the protocol but mistypes the server's name in the first message he enters into the platform. The role specification capturing this error looks as follows:

$evoting\_symbol(H) =$
$(\{\langle\langle \text{'D'},D\rangle, \langle \text{'P'},P\rangle, \langle \text{'S'},S\rangle, \langle \text{'vote'},vote\rangle, \langle \text{'birthday'},birthday\rangle\rangle\},$
$[\ In\_S_H(\langle \text{'D'},D\rangle, \langle \text{'H'},H\rangle, \langle\langle \text{'ID'},ID\rangle, \langle \text{'symbol'},symbol\rangle, \langle \text{'code'},code\rangle,$
$\langle \text{'barcode'},barcode\rangle\rangle),$
$Out\_S_H(To(\langle \text{'S'},S\rangle), \langle \text{'H'},H\rangle, \langle \text{'P'},P\rangle, \langle\langle \text{'S'}, S_{typo}\rangle, \langle \text{'ID'},ID\rangle, \langle \text{'vote'},vote\rangle\rangle),$
$In\_S_H(From(\langle \text{'S'},S\rangle), \langle \text{'P'},P\rangle, \langle \text{'H'},H\rangle,$
$captcha(\langle \text{'vote'},vote\rangle, \langle \text{'symbol'},symbol\rangle)),$
$Out\_S_H(\langle \text{'H'},H\rangle, \langle \text{'P'},P\rangle, \langle\langle \text{'birthday'},birthday\rangle, \langle \text{'code'},code\rangle\rangle),$
$In\_S_H(\langle \text{'P'},P\rangle, \langle \text{'H'},H\rangle, \langle\langle \text{'barcode'},barcode\rangle, \langle \text{'symbol'},symbol\rangle\rangle),$
$Claim_H(secret, \langle \text{'vote'},vote\rangle),$
$Claim_H(authentic\_hisp\_agreement, \langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, \langle \text{'vote'},vote\rangle),$
$Claim_H(agreement\_message, \langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, \langle \text{'vote'},vote\rangle)\ ]\ ).$

Note that the human still intends to communicate with the role S named *'S'* that he has in his initial knowledge, but only enters the wrong value for it in the second event. We model the possibility of an error by allowing the adversary to choose between the original human role specification and this one. The analysis with Tamarin shows that the protocol is robust against such a mistyping mistake of the human. Even if the ground assumptions $NoOverwrite$ and *NoTell('symbol')* are not included, all claims are shown to be satisfied. This means that the corresponding claim events are only reached in protocols which fulfill the desired properties. This is because the agents abort the protocol if they do not receive the expected messages.

On the example of the protocol *evoting_symbol* we have shown how protocols can be evaluated with respect to different assumptions with our model. We have further demonstrated how the model can be used to test whether a protocol is robust against a particular human error instead of all possible errors.
Next, we study the protocol *SmsEbanking* and present a way of making protocols more robust against role substitution attacks.

8.2 *SmsEbanking*

In the second case study, we model an e-banking protocol. We assume that a human H has access to a platform P through which he can communicate with the bank's server S. H has a device D, for example a mobile phone, to which S has a confidential channel. The channel from H to P is assumed to be confidential, the channel from D to H secure and the channel from the platform P to the server S insecure. P and S do, however, share a symmetric key.
Again, we first introduce the protocol and then analyze it. After that, we show as an alternative countermeasure, how the original protocol can be changed to be more robust. The new version then requires less knowledge of the human to achieve the same security properties.

Figure 10: Protocol *SmsEbanking*

0.	H:	knows($\langle P, D, S\rangle$)
0.	D:	knows($\langle H\rangle$)
0.	P:	knows($\langle H, S, k(P, S)\rangle$)
0.	S:	knows($\langle H, D, P, k(P, S)\rangle$)
1.	H $\circ\!\!\rightarrow\!\!\bullet$ P:	$fresh(m).\langle S, H, m\rangle$
2.	P $\circ\!\!\rightarrow\!\!\circ$ S:	$\{H, m\}_{k(P,S)}$
3.	S $\circ\!\!\rightarrow\!\!\bullet$ D:	$fresh(c).\langle c, m\rangle$
4.	D $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle c, m\rangle$
5.	H $\circ\!\!\rightarrow\!\!\bullet$ P:	c
6.	P $\circ\!\!\rightarrow\!\!\circ$ S:	$\{H, m, c\}_{k(P,S)}$

8.2.1 Protocol Specification

The Alice&Bob notation of the protocol is depicted in Figure 10. First, the human H produces a fresh message m, for example an instruction he wants to enter, which he sends to the platform P, together with S and H. P encrypts H together with m with the symmetric key he is sharing with S and sends the resulting message to S. Upon receiving this message, S produces a fresh code c which he sends, together with m, confidentially to D. This models for example that the server sends a SMS to the human's mobile phone D. The pair consisting of code and message is transferred from D to H, who then enters the code c into the platform. Finally, platform P sends an encryption of H, the message m, and the code c back to the server S.
All role specifications describing the protocol are presented in the following:

$$
\begin{aligned}
&\mathit{SmsEbanking}(P) = \\
&\quad (\{\langle\langle\text{'}H\text{'},H\rangle, \langle\text{'}S\text{'},S\rangle, k(\langle\text{'}P\text{'},P\rangle, \langle\text{'}S\text{'},S\rangle)\rangle\}, \\
&\quad [\, \mathit{In\_C}_P(\langle\text{'}H\text{'},H\rangle, \langle\text{'}P\text{'},P\rangle, \langle\langle\text{'}S\text{'},S\rangle, \langle\text{'}H\text{'},H\rangle, \langle\text{'}message\text{'},m\rangle\rangle), \\
&\quad \mathit{Out}_P(\{\langle\text{'}H\text{'},H\rangle, \langle\text{'}message\text{'},m\rangle\}_{k(\langle\text{'}P\text{'},P\rangle,\langle\text{'}S\text{'},S\rangle)}), \\
&\quad \mathit{In\_C}_P(\langle\text{'}H\text{'},H\rangle, \langle\text{'}P\text{'},P\rangle, \langle\text{'}code\text{'},c\rangle) \\
&\quad \mathit{Out}_P(\{\langle\text{'}H\text{'},H\rangle, \langle\text{'}message\text{'},m\rangle, \langle\text{'}code\text{'},c\rangle\}_{k(\langle\text{'}P\text{'},P\rangle,\langle\text{'}S\text{'},S\rangle)})\ ]\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SmsEbanking}(D) = \\
&\quad (\{\langle\text{'}H\text{'},H\rangle\}, \\
&\quad [\, \mathit{In\_C}_D(\langle\text{'}S\text{'},S\rangle, \langle\text{'}D\text{'},D\rangle, \langle\langle\text{'}code\text{'},c\rangle, \langle\text{'}message\text{'},m\rangle\rangle), \\
&\quad \mathit{Out\_S}_D(\langle\text{'}D\text{'},D\rangle, \langle\text{'}H\text{'},H\rangle, \langle\langle\text{'}code\text{'},c\rangle, \langle\text{'}message\text{'},m\rangle\rangle)\ ]\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SmsEbanking}(S) = \\
&\quad (\{\langle\langle\text{'}H\text{'},H\rangle, \langle\text{'}D\text{'},D\rangle, \langle\text{'}P\text{'},P\rangle, k(\langle\text{'}P\text{'},P\rangle, \langle\text{'}S\text{'},S\rangle)\rangle\}, \\
&\quad [\, \mathit{In}_S(\mathit{From}(\langle\text{'}H\text{'},H\rangle), \{\langle\text{'}H\text{'},H\rangle, \langle\text{'}message\text{'},m\rangle\}_{k(\langle\text{'}P\text{'},P\rangle,\langle\text{'}S\text{'},S\rangle)}), \\
&\quad \mathit{Fresh}_S(\langle\text{'}code\text{'},c\rangle), \\
&\quad \mathit{Out\_C}_S(\langle\text{'}S\text{'},S\rangle, \langle\text{'}D\text{'},D\rangle, \langle\langle\text{'}code\text{'},c\rangle, \langle\text{'}message\text{'},m\rangle\rangle), \\
&\quad \mathit{In}_S(\{\langle\text{'}H\text{'},H\rangle, \langle\text{'}message\text{'},m\rangle, \langle\text{'}code\text{'},c\rangle\}_{k(\langle\text{'}P\text{'},P\rangle,\langle\text{'}S\text{'},S\rangle)}), \\
&\quad \mathit{Claim}_S(\mathit{authentic\_hisp\_agreement}, \langle\text{'}H\text{'},H\rangle, \langle\text{'}S\text{'},S\rangle, \langle\text{'}message\text{'},m\rangle)\ ]\ )
\end{aligned}
$$

Figure 11: Countermeasure in Protocol *SmsEbanking_fix*

$NoGet(\textit{`code'})$ $NoTell(\textit{`message'})$ $NoGet(\textit{`message'})$		
H $\circ\!\!\rightarrow\!\!\bullet$ R1:	$\langle\langle yn,y\rangle, \langle \textit{`H'},H\rangle, \langle \textit{`message'},m\rangle\rangle$	$to : yn, y$
R2 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle \textit{`code'},c\rangle, \langle \textit{`message'},m\rangle\rangle$	

$$
\begin{aligned}
&SmsEbanking(H) = \\
&\quad (\{\langle\langle \textit{`P'},P\rangle, \langle \textit{`D'},D\rangle, \langle \textit{`S'},S\rangle\rangle\}, \\
&\quad [\, Fresh_H(\langle \textit{`message'},m\rangle), \\
&\quad Out\_C_H(To(\langle \textit{`S'},S\rangle), \langle \textit{`H'},H\rangle, \langle \textit{`P'},P\rangle, \langle\langle \textit{`S'},S\rangle, \langle \textit{`H'},H\rangle, \langle \textit{`message'},m\rangle), \\
&\quad In\_S_H(\langle \textit{`D'},D\rangle, \langle \textit{`H'},H\rangle, \langle\langle \textit{`code'},c\rangle, \langle \textit{`message'},m\rangle\rangle), \\
&\quad Out\_C_H(\langle \textit{`H'},H\rangle, \langle \textit{`P'},P\rangle, \langle \textit{`code'},c\rangle) \,]\,).
\end{aligned}
$$

The *authentic_hisp_agreement* claim of S shows the goal of the protocol, namely that the server S wants to be sure that H intentionally sends $\langle \textit{`message'},m\rangle$ to S.

8.2.2 Analysis

We have analyzed the protocol with Tamarin, with respect to the original human role specification as well as with respect to all human role specifications the adversary can produce. Further, we examined the protocol *SmsEbanking_fix*, which takes into account the countermeasure depicted in Figure 11. The countermeasure describes that the human knows he can only receive *'code'* and *'message'* as well as send *'message'* by the exceptions captured in the known steps. The first known step describes that he confidentially sends a concatenation of an intended receiver $\langle yn,y\rangle$, his own role $\langle \textit{`H'},H\rangle$ and of a message $\langle \textit{`message'},m\rangle$. The second known step says that he can then receive, over a secure channel, a message $\langle \textit{`code'},c\rangle$ together with the same message $\langle \textit{`message'},m\rangle$ that he has previously sent as specified by the first known step. Expressing the countermeasure formally, results in:

$$
\begin{aligned}
&K(H) = \\
&\quad \{NoGet(\textit{`code'}), NoTell(\textit{`message'}), NoGet(\textit{`message'})\}, \\
&\quad [\, Out\_C_H(To(\langle yn,y\rangle), \langle Hn,H\rangle, \langle R1n,R1\rangle, \langle\langle yn,y\rangle, \langle \textit{`H'},H\rangle, \langle \textit{`message'},m\rangle\rangle), \\
&\quad In\_S_H(\langle R2n,R2\rangle, \langle Hn,H\rangle, \langle\langle \textit{`code'},c\rangle, \langle \textit{`message'},m\rangle\rangle) \,]\,).
\end{aligned}
$$

In addition to the claimed property *authentic_hisp_agreement*, the weaker authentication properties *alive*, *recent_alive*, *authentic_hisp* and *authentic_hisp_withName* have been analyzed. The results of the analysis are depicted in Table 3 and the symbols can be interpreted as in the last section.

With the original human role specification ('orig. spec'), all tested authentication claims of S are satisfied. If the human has no knowledge ('all spec.'), however, even the weakest authentication claim *alive* is falsified by Tamarin.

Table 3: Overview Findings Protocol *SmsEbanking*

	SmsEbanking		*SmsEbanking_fix*
	orig. spec.	all spec.	
alive	✓	×	(✓)
recent_alive	✓	(×)	(✓)
authentic_hisp	✓	(×)	(✓)
authentic_hisp_withName	✓	(×)	(✓)
authentic_hisp_agreement	✓	(×)	✓

The attack against this claim is depicted in Figure 12. First, a human *H1* which is not the one claimed to be alive, receives from the adversary the message $\langle$*'H',H*$\rangle$ and produces $\langle$*'message',m*$\rangle$ freshly. Not knowing that he should send a message containing his own identity, he sends the two messages together with $\langle$*'S',S*$\rangle$ to the platform P. P encrypts the received $\langle$*'H',H*$\rangle$ and $\langle$*'message',m*$\rangle$ and sends the result to S. The server next produces a fresh code $\langle$*'code',c*$\rangle$ and sends it together with the message $\langle$*'message',m*$\rangle$ to the device D which he thinks belongs to H. The information on device D is, however, read by the human *H1*, who forwards $\langle$*'code',c*$\rangle$ to the platform P, from which it is sent back to S encrypted together with $\langle$*'message',m*$\rangle$ and $\langle$*'H',H*$\rangle$. S wrongly concludes that H is alive, when actually the human *H1* was participating in the protocol. The role specification which is instantiated by the agent *H1* and leads to this attack is shown in the following. Note that no role specification for H needs to be specified, because he was never doing anything.

$$
\begin{aligned}
&SmsEbanking(H) = \\
&\quad (\{\langle\langle 'P',P\rangle, \langle 'D',D\rangle, \langle 'S',S\rangle\rangle\}, \\
&\quad [\; In_H(\langle 'H',H2\rangle), \\
&\quad Fresh_H(\langle 'message',m\rangle), \\
&\quad Out\_C_H(\langle 'H',H\rangle, \langle 'P',P2\rangle, \langle\langle 'S',S\rangle, \langle 'H',H2\rangle, \langle 'message',m\rangle\rangle), \\
&\quad In\_S_H(\langle 'D',D2\rangle, \langle 'H',H\rangle, \langle\langle 'code',c\rangle, \langle 'message',m\rangle\rangle) \;]\;) \\
&\quad Out\_C_H(\langle 'H',H\rangle, \langle 'P',P2\rangle, \langle 'code',c\rangle) \;]\;)
\end{aligned}
$$

With Tamarin, the protocol *SmsEbanking_fix* is shown to satisfy all claimed properties.

8.2.3 Variation: Robuster Protocol Specification

In this section, we show on the example of the protocol *SmsEbanking*, that it is sometimes possible to make protocols more robust by making the agents exchange more messages. The idea is that if they exchange more information, they will thwart errors in the human role specification that were leading to attacks before.

Figure 13 shows protocol *SmsEbanking_robuster*, which is a robuster version of the introduced e-banking protocol. The new messages compared to the original protocol, *SmsEbanking*, are depicted in bold. In the third step, S additionally sends to D the agent names S and H, and D also sends S to H in the fourth

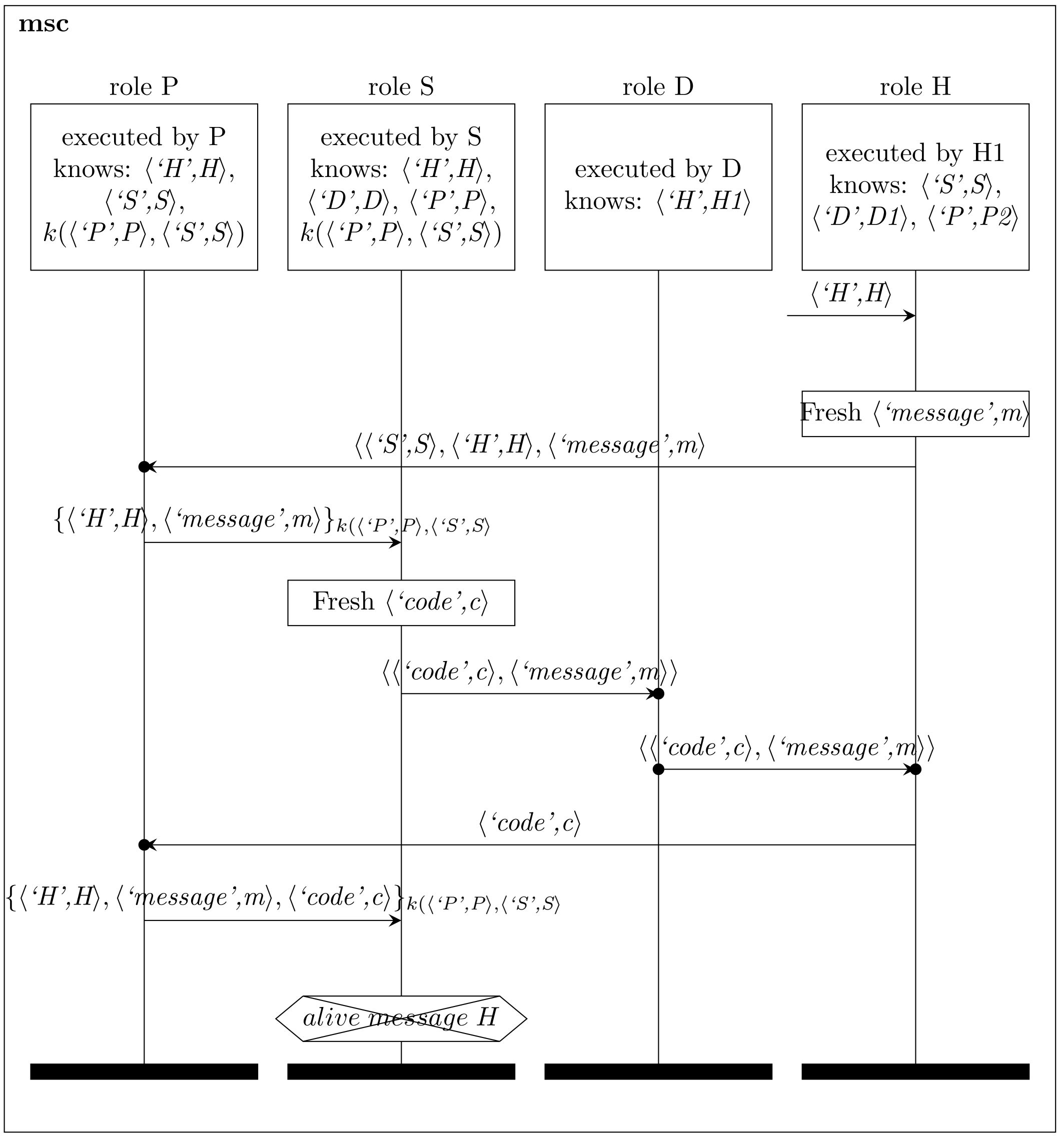

msc
role P
role S
role D
role H
executed by P knows: ⟨'H',H⟩, ⟨'S',S⟩, k(⟨'P',P⟩, ⟨'S',S⟩)
executed by S knows: ⟨'H',H⟩, ⟨'D',D⟩, ⟨'P',P⟩, k(⟨'P',P⟩, ⟨'S',S⟩)
executed by D knows: ⟨'H',H1⟩
executed by H1 knows: ⟨'S',S⟩, ⟨'D',D1⟩, ⟨'P',P2⟩
⟨'H',H⟩
Fresh ⟨'message',m⟩
⟨⟨'S',S⟩, ⟨'H',H⟩, ⟨'message',m⟩
{⟨'H',H⟩, ⟨'message',m⟩}k(⟨'P',P⟩,⟨'S',S⟩)
Fresh ⟨'code',c⟩
⟨⟨'code',c⟩, ⟨'message',m⟩⟩
⟨⟨'code',c⟩, ⟨'message',m⟩⟩
⟨'code',c⟩
{⟨'H',H⟩, ⟨'message',m⟩, ⟨'code',c⟩}k(⟨'P',P⟩,⟨'S',S⟩)
alive message H

Figure 12: Protocol *SmsEbanking*: attack on the claim $Claim_S(alive, \langle\text{'}H\text{'},H\rangle, \langle\text{'}S\text{'},S\rangle, \langle\text{'}message\text{'},m\rangle)$ of S

Figure 13: Protocol *SmsEbanking_robuster*

0.	H:	knows($\langle P, D, S\rangle$)
0.	D:	knows($\langle H\rangle$)
0.	P:	knows($\langle H, S, k(P, S)\rangle$)
0.	S:	knows($\langle H, D, P, k(P, S)\rangle$)
1.	H $\circ\!\!\rightarrow\!\bullet$ P:	$fresh(m).\langle S, H, m\rangle$
2.	P $\circ\!\!\rightarrow\!\circ$ S:	$\{H, m\}_{k(P,S)}$
3.	S $\circ\!\!\rightarrow\!\bullet$ D:	$fresh(c).\langle \boldsymbol{S,H,}c, m\rangle$
4.	D $\bullet\!\!\rightarrow\!\bullet$ H:	$\langle \boldsymbol{S,}c, m\rangle$
5.	H $\circ\!\!\rightarrow\!\bullet$ P:	c
6.	P $\circ\!\!\rightarrow\!\circ$ S:	$\{H, m, c\}_{k(P,S)}$

step.
The following are the new role specifications for the roles D, S and H. We do not repeat the role specification of P, because it remains the same as in the original protocol.

$SmsEbanking(D) =$
$(\{\langle \text{'H'},H\rangle\},$
$[\ In\_C_D(\langle \text{'S'},S\rangle, \langle \text{'D'},D\rangle, \langle\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, \langle \text{'code'},c\rangle, \langle \text{'message'},m\rangle\rangle),$
$Out\_S_D(\langle \text{'D'},D\rangle, \langle \text{'H'},H\rangle, \langle\langle \text{'S'},S\rangle, \langle \text{'code'},c\rangle, \langle \text{'message'},m\rangle\rangle)\ ]\ )$

$SmsEbanking(S) =$
$(\{\langle\langle \text{'H'},H\rangle, \langle \text{'D'},D\rangle, \langle \text{'P'},P\rangle, k(\langle \text{'P'},P\rangle, \langle \text{'S'},S\rangle)\rangle\},$
$[\ In_S(From(\langle \text{'H'},H\rangle), \{\langle \text{'H'},H\rangle, \langle \text{'message'},m\rangle\}_{k(\langle \text{'P'},P\rangle,\langle \text{'S'},S\rangle)}),$
$Fresh_S(\langle \text{'code'},c\rangle),$
$Out\_C_S(\langle \text{'S'},S\rangle, \langle \text{'D'},D\rangle, \langle\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, \langle \text{'code'},c\rangle, \langle \text{'message'},m\rangle\rangle),$
$In_S(\{\langle \text{'H'},H\rangle, \langle \text{'message'},m\rangle, \langle \text{'code'},c\rangle\}_{k(\langle \text{'P'},P\rangle,\langle \text{'S'},S\rangle)}),$
$Claim_S(authentic\_hisp\_agreement, \langle \text{'H'},H\rangle, \langle \text{'S'},S\rangle, \langle \text{'message'},m\rangle)\ ]\ )$

$SmsEbanking(H) =$
$(\{\langle\langle \text{'P'},P\rangle, \langle \text{'D'},D\rangle, \langle \text{'S'},S\rangle\rangle\},$
$[\ Fresh_H(\langle \text{'message'},m\rangle),$
$Out\_C_H(To(\langle \text{'S'},S\rangle), \langle \text{'H'},H\rangle, \langle \text{'P'},P\rangle, \langle\langle \text{'S'},S\rangle, \langle \text{'H'},H\rangle, \langle \text{'message'},m\rangle\rangle),$
$In\_S_H(\langle \text{'D'},D\rangle, \langle \text{'H'},H\rangle, \langle\langle \text{'S'},S\rangle, \langle \text{'code'},c\rangle, \langle \text{'message'},m\rangle\rangle),$
$Out\_C_H(\langle \text{'H'},H\rangle, \langle \text{'P'},P\rangle, \langle \text{'code'},c\rangle)\ ]\ )$

We argue that this version of the protocol is more robust by presenting a countermeasure which is not sufficient to avoid attacks in the original protocol version. At the same time, the same security properties as tested before are all satisfied in the new version when the same countermeasure is taken into account. We conclude that the new protocol is more robust, because the human needs to have less knowledge to guarantee the tested security properties.
The countermeasure is captured in Figure 14 and we call the robuster version

Figure 14: Countermeasure in Protocol *SmsEbanking_robuster_fix*

$NoGet(\text{‘message’})$ $NoTell(\text{‘message’})$		
H $\circ\!\!\rightarrow\!\!\bullet$ R1:	$\langle\langle yn,y\rangle, \langle xn,x\rangle, \langle \text{‘message’},m\rangle\rangle$	$to : yn, y$
R2 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle vn,v\rangle, \langle un,u\rangle, \langle \text{‘message’},m\rangle\rangle$	

Figure 15: Countermeasure in Protocol *SmsEbanking_as_robuster_fix*

$NoGet(\text{‘message’})$ $NoTell(\text{‘message’})$		
H $\circ\!\!\rightarrow\!\!\bullet$ R1:	$\langle\langle yn,y\rangle, \langle xn,x\rangle, \langle \text{‘message’},m\rangle\rangle$	$to : yn, y$
R2 $\bullet\!\!\rightarrow\!\!\bullet$ H:	$\langle\langle un,u\rangle, \langle \text{‘message’},m\rangle\rangle$	

of the protocol which also takes this countermeasure into account *SmsEbanking_robuster_fix*. Compared to the countermeasure in the previous subsection, the *NoGet(‘code’)* is not included and the human does not need to know the names of the exchanged messages, except for the message named *‘message’*. Formally, the countermeasure looks as follows:

$$\begin{aligned}
&K(H) = \\
&\quad (\{NoTell(\text{‘message’}), NoGet(\text{‘message’})\}, \\
&\quad [\ Out\_C_H(To(\langle yn,y\rangle), \langle Hn,H\rangle, \langle R1n,R1\rangle, \langle\langle yn,y\rangle, \langle xn,x\rangle, \langle \text{‘message’},m\rangle\rangle), \\
&\quad In\_S_H(\langle R2n,R2\rangle, \langle Hn,H\rangle, \langle\langle vn,v\rangle, \langle un,u\rangle, \langle \text{‘message’},m\rangle\rangle)\ ]\ )
\end{aligned}$$

To compare the new version with the original one, we analyze the original protocol with respect to the same countermeasure idea, which we call *SmsEbanking_as_robuster_fix*. Because of the difference in the protocol specification, we adjust the second step of the countermeasure to only contain two messages, which is shown in Figure 15.

An overview of the evaluation of both protocols with Tamarin is shown in Table 4. While all tested properties are satisfied by *SmsEbanking_robuster_fix*, not

Table 4: Overview Findings Protocols *SmsEbanking_robuster_fix* and *SmsEbanking_as_robuster_fix*

	SmsEbanking _robuster_fix	*SmsEbanking _as_robuster_fix*
alive	(✓)	×
recent_alive	(✓)	(×)
authentic_hisp	(✓)	(×)
authentic_hisp_withName	(✓)	(×)
authentic_hisp_agreement	✓	(×)

even *alive* is given in *SmsEbanking_as_robuster_fix*. In fact, the same attack which was in the previous subsection shown to violate *alive* in the protocol without any countermeasures is still possible. In contrast to this, the additional exchanged information in *SmsEbanking_robuster* makes sure that S and D agree on who is in the role of H, which makes this attack impossible.

We have demonstrated that sometimes more robust versions of a protocol can be found by adding more information to the message exchange between the agents.

9 Hierarchies of Protocols

In this section we want to present hierarchies to compare how much a human has to know in different protocols for certain security properties to hold. Because the knowledge of a human in a given protocol is described by the countermeasure, we need a notion of comparing the amount of knowledge captured by each countermeasure. It is difficult to decide what kind of knowledge is harder to remember for humans and it might additionally depend on the opinion of the individual human who participates in a protocol. One human might find it easier to remember a lot of general rules saying that he can never communicate certain things, while another might find it easier to remember concrete steps that have to happen before others. We leave it open how the knowledge of different countermeasures is best compared and present different ideas together with the respective resulting hierarchies of protocols. For a better overview, we only capture the security properties claimed from one agent in one hierarchy. We first describe how the hierarchies are to be interpreted in general and then introduce one possible hierarchy with respect to the claims made by the server role S in human-server communication. Then, we present two hierarchies which are more fine grained but built upon the first one. Finally, we consider one of the presented hierarchies with respect to the claims made by the human H.

In all of the hierarchies, depicted in Figures 16 to 19, all the protocols [17] are analyzed with the Tamarin tool and under the assumption that an adversary does not compromise participating agents. The protocols are evaluated with respect to the countermeasure described in the corresponding specification file or with respect to a human without knowledge if none is specified. We use the color yellow to denote that secrecy is given in the corresponding protocol, blue to denote that authentication is given, and green to denote that both secrecy and authentication of the claimed term are given. Thereby, we use light, medium or dark blue to differentiate if the authentication claim *alive*, *authentic_hisp*, or *authentic_hisp_agreement* is satisfied. Similarly, we use medium and dark green to denote that *secrecy* and *authentic_hisp* or *secrecy* and *authentic_hisp_agreement* are satisfied. Further, we underline the protocol name to denote that *agreement_message* is given. If a protocol is uncolored, only blue, only yellow, or not underlined it either means that stronger claims are not fulfilled or that stronger claims have not been tested from the point of view of the agent considered in this hierarchy. For example if a protocol is dark blue and satisfies *authetic_hisp_agreement* with respect to a certain message, either *secrecy* was tested and did not hold or it was not tested because it is not a goal of the protocol an we, therefore, do not care whether it holds or not.

Whenever claims are made with respect to more than one message, the color of the protocol reflects the strongest property which is given for all of them.
We create a hierarchy by starting with protocols containing no human knowledge at the top and by ending with a group containing all considered kinds of knowledge at the bottom. Thereby, a connection from one group to a lower group denotes that the lower group contains more knowledge. We say that a group with less knowledge is an ancestor of a group with more knowledge, if all the knowledge from the first group is also included in the second group. Consequently, every group would also be connected to all ancestors of its ancestors. For better readability, we only connect the direct ancestors. Further, only the defined groups which contain a considered protocol are shown in the hierarchies.

We first present a hierarchy of the protocols with respect to the claims of role S, depicted in Figure 16. The group denoted by nK consists of the protocols where the human has no knowledge. Then, we distinguish between countermeasures that contain a $NoTell$, a $NoGet$, a $NoOverwrite$, a known step, or a combination of these. We add to the respective group name nT, nG, nO, KnS, or the appropriate combination. Some of the protocol goals could be achieved by either including $NoGet$ or $NoOverwrite$ in the countermeasure. We chose to use $NoGet$ whenever we saw a possibility, to make the different protocols as comparable as possible.
From the hierarchy it can for example be seen that none of the protocols in the group nK, where there is no human knowledge, satisfy *secrecy* or *authentic_hisp_agreement*. At the same time it can be seen that there are protocols which already satisfy *authentic_hisp* without any human knowledge. It can further be observed that in all the examined protocols, secrecy is only satisfied if the human knows at least $NoTell$ and *authentic_hisp_agreement* if the human has at least $NoTell$ and a known step in his knowledge.

As a next step, we extend the hierarchy by assuming that more known steps or more noDo rules in the countermeasure require a bigger human knowledge. For this reason, we count the occurrences of nT, nG, nO and KnS. The resulting hierarchy is shown in Figure 17. Several claim steps are counted as one step, because for each security property to hold, only the corresponding claim has to be done. Further, if $NoTell$, $NoGet$ and $NoOverwrite$ are used without arguments they are counted as one occurrence of nT, nG and nO.
Compared to the previous hierarchy, a more fine grained distinction of how much knowledge is needed can be made. The protocols *ChangePwd_fix* and *vote_icon_fixAll* are in the same group of the previously presented hierarchy. Now it can be seen that *vote_icon_fixAll* requires more knowledge with respect to this hierarchy, because two $NoTell$, one $NoGet$ and three known steps are needed to achieve the same security properties that are satisfied in *ChangePwd_fix* with only two $NoTell$, one $NoGet$ and two known steps. The observation that in the examined protocols *secrecy* is only satisfied if the human knows at least $NoTell$ can still be made.

As a third idea of how to define a hierarchy we show another more fine grained version of the first hierarchy. Instead of counting the occurrences of the countermeasures distinguished in the first hierarchy, we make a more detailed distinction of the known steps appearing in the countermeasures. We distinguish between

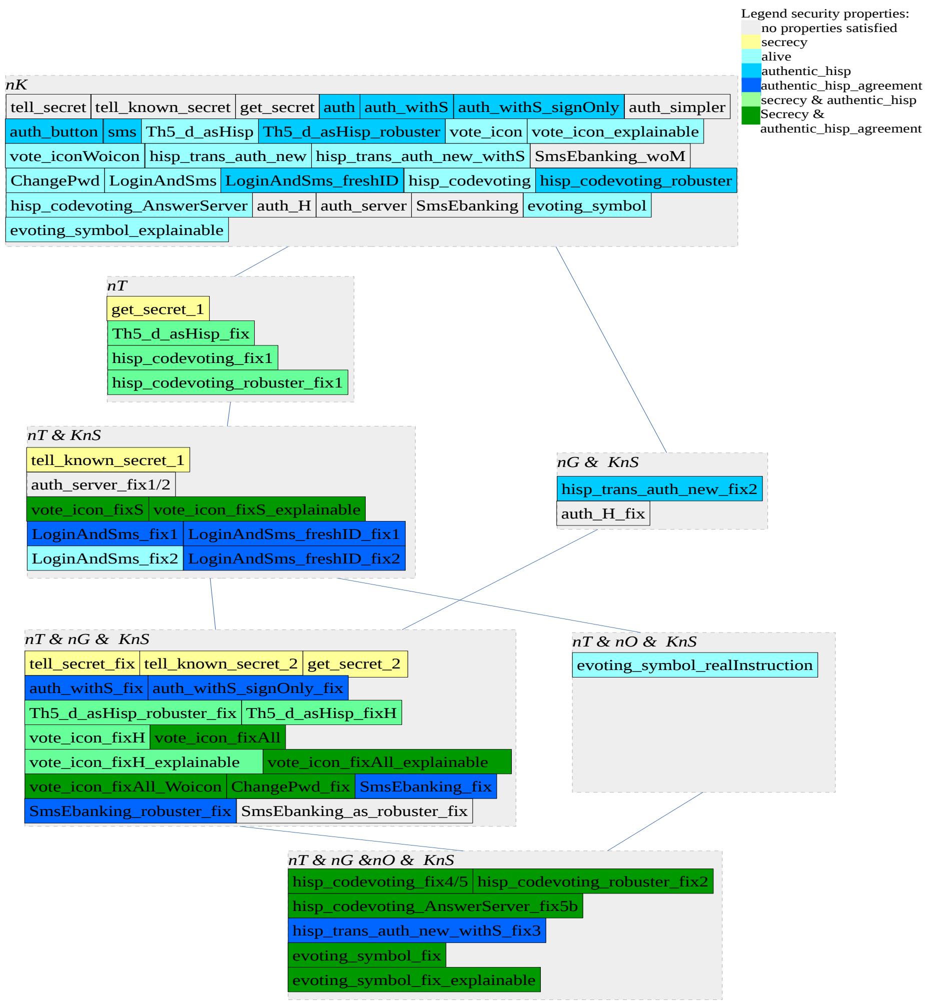
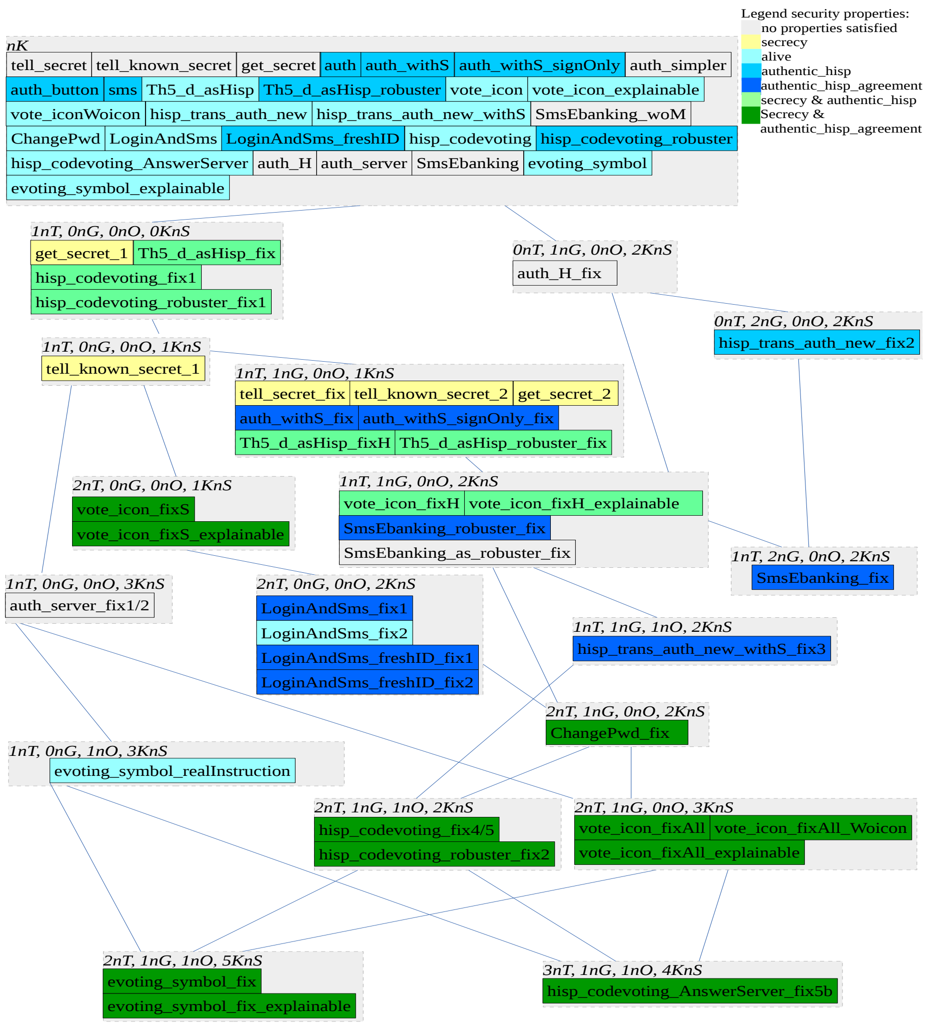


Figure 16: Hierarchy of countermeasures required to satisfy the claims of the server role in human-server communication. The hierarchy considers $NoTell$ rules (nT), $NoGet$ rules (nG), $NoOverwrite$ rules (nO) and known steps (KnS).

Figure 17: Hierarchy of countermeasures required to satisfy the claims of the server role in human-server communication. The hierarchy counts the occurrences of $NoTell$ rules (nT), $NoGet$ rules (nG), $NoOverwrite$ rules (nO) and known steps (KnS).



known steps with an intention, known steps containing claims and simple known steps that do not contain an intention or claim. The respective properties are denoted by Int, Cl and KnS in the hierarchy. We then say knowing that there is an intention or claim in a known step requires more knowledge than just knowing a simple step. For this reason, the countermeasures with simple known steps KnS are ancestors of the countermeasures with known steps containing intentions or claims. At the same time, countermeasures with Int or Cl only are still ancestors of countermeasures where Int and KnS or Cl and KnS are needed. The hierarchy capturing this idea is depicted in Figure 18. Again, compared to the first hierarchy it is possible to see a more fine grained distinction of which protocols need more knowledge. We compare again the protocols *ChangePwd_fix* and *vote_icon_fixAll*. We learn that *vote_icon_fixAll* also requires more knowledge with respect to this hierarchy; in addition to the knowledge required for *ChangePwd_fix*, a claim needs to be known. Recall the observation of the first hierarchy that, in the examined protocols, at least a $NoTell$ and a known step are required in the countermeasure for *authentic_hisp_agreement* to hold. This statement can now be made more precise by observing in this hierarchy that, in all considered protocols, at least a known step containing an intention and a $NoTell$ have to be in the human's knowledge for *authentic_hisp_agreement* to hold.

Similarly, many other hierarchies could be imagined. All of them could have a different definition of what a human perceives to be a more complicated countermeasure to remember. Every hierarchy can then also be examined with respect to claims of different roles in which case the considered countermeasures in the protocols are the same, so the groups of the hierarchies remain unchanged. The only thing that changes are the security properties that are satisfied from the perspective of another role, that is the colors of the protocols in the hierarchy. As an example, we show the last introduced hierarchy again in Figure 19 with respect to the claims of the human role H. It can be seen that in all considered protocols without human knowledge no security properties claimed by the human are given. Further, in all examined protocols at least $NoTell$ and $NoGet$ have to be given in order for *secrecy* to be satisfied. It can also be observed that *authentic_hisp_agreement* is only given in the considered protocols if the human knows at least Int, Cl, and either $NoTell$ or $NoGet$.

We have shown in this section how hierarchies can be defined to compare the human knowledge in different protocols and to see what security properties are satisfied by each protocol. Hierarchies can be useful to find general observations, as for example that in the examined security protocols, secrecy is only given if there is at least a $NoTell$ rule in the countermeasure. Observations like this could help to find in future work design principles which state what facts have at least to be communicated to the human such that certain security properties can be satisfied.

10 Future Work

Because this thesis is a first attempt to model human errors in a formal model, we are aware of the fact that we have not covered all aspects of the problem

Figure 18: Hierarchy of countermeasures required to satisfy the claims of the server role in human-server communication. The hierarchy considers $NoTell$ rules (nT), $NoGet$ rules (nG), $NoOverwrite$ rules (nO), simple known steps (KnS), known steps with an intention (Int), and known steps with a claim (Cl).

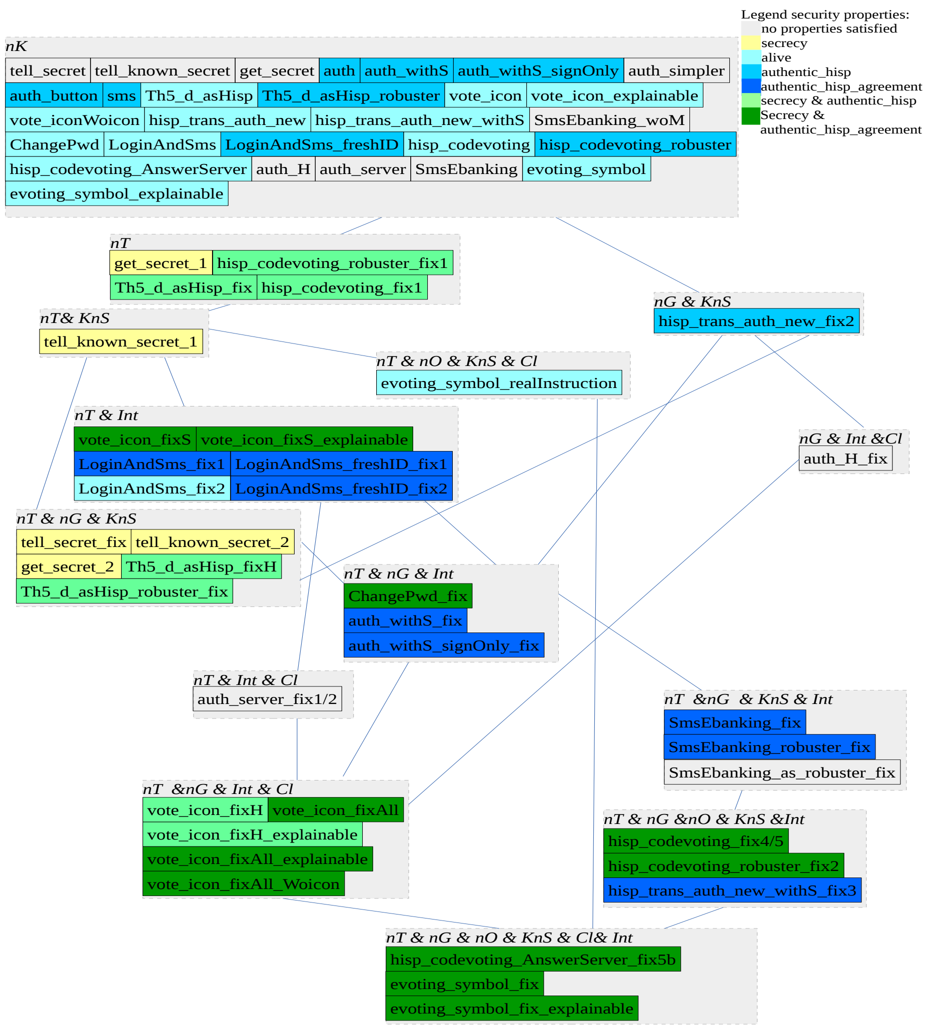
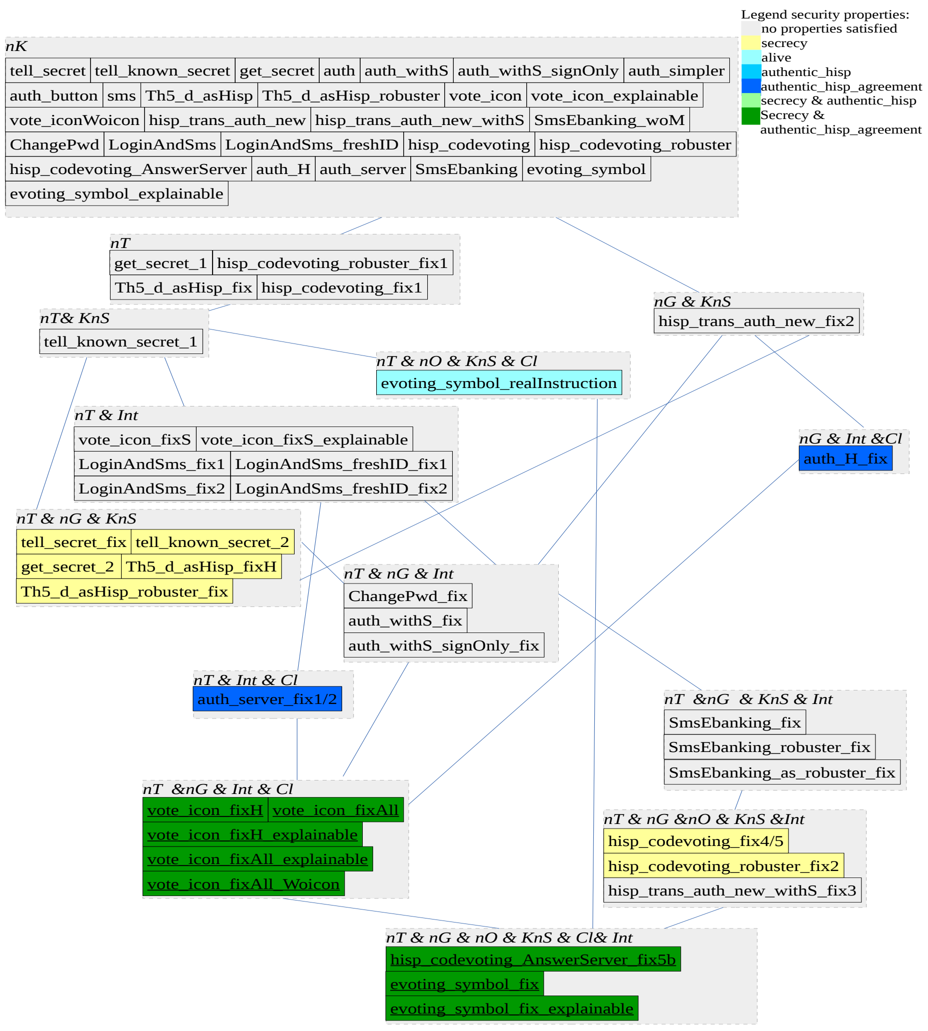


Figure 19: Hierarchy of countermeasures required to satisfy the claims of the human role in human-server communication. The hierarchy considers $NoTell$ rules (nT), $NoGet$ rules (nG), $NoOverwrite$ rules (nO), simple known steps (KnS), known steps with an intention (Int), and known steps with a claim (Cl).



or all possible countermeasures. For this reason, we dedicate this section to the presentation of some ideas that could be examined in future work. We first present two ideas how the countermeasures could be extended. Then, we show another security property that could be defined together with a new countermeasure that would be a natural consequence of it. Next, we suggest how another channel model might help analyzing the interaction between humans and devices more realistically. Finally, we say that the role substitution attack could be defined such that redundant human role specifications would not be considered.

No order of the known steps: An assumption we have made in defining the countermeasures is that if a human has several known steps in his knowledge he always knows their relative order. Alternatively, it could be examined if it is also a useful countermeasure to assume that the human has several steps in his knowledge without a defined order. The steps would then still describe exceptions to the noDo knowledge but not in what order the respective events have to happen. Another option would be to define that the human knows that certain events, for example claims, will only happen when a set of other events has happened, not knowing the order of the events in that set.

Intention in a general rule: In the countermeasures as they are now, we always specify intention in the step where it is needed. We either express the intended receiver or sender by specifying a fixed variable name or by reusing an unknown variable from elsewhere in the countermeasure.

Example 10.1. The step A $\circ\!\!\rightarrow\!\!\circ$ H: $\langle$*'message',m*$\rangle$ $from : \langle$*'S',S*$\rangle$ denotes by the name *'S'* with respect to whom the intention has to be made. In contrast, the step A $\circ\!\!\rightarrow\!\!\circ$ H: $\langle\langle$*'message',m*$\rangle, \langle$*xn,x*$\rangle\rangle$ $from : \langle$*xn,x*$\rangle$ specifies the intended sender by saying that he equals the second component of the received message.

The latter could also be expressed by a general rule stating that whenever the human receives a message, he intends to have it from the role that is defined by the last component of the received message. Alternatively, one can define that a human always receives a concatenation of messages where the first component denotes the intended sender. If such a rule is not applicable for every received message, the protocol can be adjusted such that the non-human entities send all the messages to the human in this form. It is possible to define the same for intention in sending events. Intention could then be described by a general rule, rather than in every step separately. It can be argued that this requires less knowledge of the human because he does not have to remember the details of where to find the intention in every step.

Security property *origin* and countermeasure *fresh*: Many security properties different from the introduced ones could be considered as well. We introduce the idea of a security property *origin* together with a countermeasure *fresh* for showing that also the countermeasures are possibly not completed. If an agent claims *origin* with respect to a message and a human, this means that he not only thinks that the human sent the message but also that the human produced it freshly. This property can be used to model that a human communicates his own opinion rather than repeating a message which was given to him, potentially by the adversary. Because the security property *origin* requires a trace in which the human performs a $Fresh$ event, a natural extension of the countermeasures is to allow specifying that a human knows he has to produce a message $\langle$*'message'*,$m\rangle$ freshly before sending it. This can be expressed by the known step H: $fresh(\langle$*'message'*,$m\rangle)$ in the countermeasures. Even if a $Fresh$ event could also be enforced indirectly by stating with a $NoGet$(*'message'*) that a human cannot receive $\langle$*'message'*,$m\rangle$ from anybody, this shows that other countermeasures might be more natural or could require fewer known steps to achieve the same things as the introduced countermeasures.

Human channels: In the present thesis we have considered insecure, authentic, confidential and secure communication channels. We have not distinguished channels between two non-human agents from channels between a non-human and a human agents. A human reading several messages from a device display has for example been modeled by a send event on a secure channel from the device to the human which contains a concatenation of the messages in a defined order. In future work the channels from a device to a human could be modeled under the assumption that the way a human reads the displayed information cannot be influenced by an adversary. It is still possible, that the adversary changes or blocks parts of the information that is sent to a device over the network. If, however, several messages are already on the screen of a device, it could newly be assumed that an adversary cannot influence how the human reads them. A new channel taking these assumptions into consideration could help to model the interaction between a human and a display more realistically. One channel property that could be distinguished is that the order of several communicated messages does not have to be fixed. Because the human will himself choose which information to read first from the display and will also notice if different messages are switched, it can be argued that a channel specifying the communicated information but not the order of the messages is more realistic.

Derivable Human Role Specifications: We have defined in the role substitution attack what new human role specifications can be derived by an adversary. Thereby, we did not care whether there are role specifications which are redundant or which will never help in forming attacks. It could be considered in future work how the set of derivable human role specifications can be defined such that only meaningful specifications are considered. If for example a send of the message $\langle$*'message'*,$m\rangle$ to an insecure channel is added to the human role specification, the adversary can learn this message and reuse it as he wants. For this reason, role specifications that contain two or more consecutive sends of this message will not help the adversary to learn more or to form new attacks; these role specifications can be viewed to be redundant. Further, in a protocol where the role $\langle$*'R'*,$R\rangle$ never expects to receive anything from the human over a secure channel, an event $Out\_S_H(\langle$*'H'*,$H\rangle, \langle$*'R'*,$R\rangle, \langle$*'message'*,$m\rangle)$ in the human role specification will not help to form new attacks, because the sent message can never be received by a role. Defining such restrictions would reduce the set of possible human role specifications and could enable the production of an exhaustive list of all relevant role specifications.

11 Conclusion

We have introduced a formal model to analyze human errors in security protocols. Thereby, we have assumed that a human has no knowledge about the protocol and that the adversary can exchange his role specification by a new one in a role substitution attack. Because this attack allows to express all possible deviations of the human role specification, the model is able to capture all attacks arising from human errors. We have then presented countermeasures which express that a human has some knowledge about parts of the protocol that he will do correctly. This limits what human role specifications an adversary is able to derive. As an alternative countermeasure we have shown that by changing the original protocols, it is sometimes possible to achieve more robust versions thereof which require less human knowledge to satisfy the security goals. In two case studies we have then analyzed an e-voting and an e-banking protocol with the help of the Tamarin tool. In both protocols, we have found attacks that are possible if the human has no knowledge and have then introduced countermeasures which provably prevent the attacks. Finally, we have introduced hierarchies that enable the comparison of the human knowledge in different security protocols. This could help to define in future work what countermeasures have at least to be communicated to the human such that certain security properties can be satisfied.

References

[1] *SOUPS '13: Proceedings of the Ninth Symposium on Usable Privacy and Security*, New York, NY, USA, 2013. ACM.

[2] E-voting Demo System. `http://www.demo-webvote.ch/#/anleitung-demo`, last access: March 30th 2015.

[3] Luis Von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using Hard AI Problems for Security. In *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'03, pages 294–311, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] D. Basin, S. Radomirović, and M. Schläpfer. A Complete Characterization of Secure Human-Server Communication. In *28th IEEE Computer Security Foundations Symposium (CSF 2015)*. IEEE Computer Society, 2015.

[5] David Chaum. Surevote: technical overview. In *Proceedings of the workshop on trustworthy elections (WOTE'01)*, 2001.

[6] Cas Cremers and Sjouke Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012.

[7] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[8] Gavin Lowe. A Hierarchy of Authentication Specifications. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 31–44. IEEE Computer Society, 1997.

[9] Mohammad Mannan and P. C. van Oorschot. Security and Usability: The Gap in Real-world Online Banking. In *Proceedings of the 2007 Workshop on New Security Paradigms*, NSPW '07, pages 1–14, New York, NY, USA, 2008. ACM.

[10] Ueli Maurer and Pierre Schmid. A Calculus for Secure Channel Establishment in Open Networks. In Dieter Gollmann, editor, *European Symposium on Research in Computer Security — ESORICS '94*, volume 875 of *Lecture Notes in Computer Science*, pages 175–192. Springer-Verlag, November 1994. Final version: [11].

[11] Ueli Maurer and Pierre Schmid. A Calculus for Security Bootstrapping in Distributed Systems. *Journal of Computer Security*, 4(1):55–80, 1996. Preliminary version: [10].

[12] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The Tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, pages 696–701. Springer, 2013.

[13] Rachna Dhamija Rachna and JD Tygar. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Citeseer, 2006.

[14] Jens Rasmussen. The role of hierarchical knowledge representation in decisionmaking and system management. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(2):234–243, 1985.

[15] James Reason. Human error: models and management. *Bmj*, 320:768–770, 2000.

[16] Michaël Rusinowitch and Mathieu Turuani. Protocol Insecurity with Finite Number of Sessions is NP-Complete. In *14th IEEE Computer Security Foundations Workshop (CSFW 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, page 174. IEEE Computer Society, 2001.

[17] Lara Schmid. Tamarin Specification Files. `http://www.infsec.ethz.ch/education/studentProjects/former/lschmid.html`.

A Declaration of originality

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Human Errors in Secure Communication Protocols

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schmid	Lara Alexandra

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Rüfenach, 04/19/2015	L Schmid

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.